Eero Laukkanen

Java source code generation from OPC UA information models

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 9.9.2013

Thesis supervisor:

D.Sc. (Tech.) Ilkka Seilonen

Thesis advisor:

M.Sc. (Tech.) Jouni Aro

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Eero Laukkanen

Title: Java source code generation from OPC UA information models

Date: 9.9.2013 Language: English Number of pages:7+57

Department of Automation and Systems Technology

Professorship: Information and Computer Systems in Automation Code: AS-116

Supervisor: D.Sc. (Tech.) Ilkka Seilonen

Advisor: M.Sc. (Tech.) Jouni Aro

OPC Unified Architecture is an industrial communication specification that introduces information modeling capabilities. These capabilities allow modeling the communicated data with an object model similar to object-oriented programming languages. However, using the information modeling capabilities is not developer-friendly in the current state of Prosys OPC UA Java SDK.

In this thesis, it is identified how the usage of information models could be made easier. First, requirements for source code generation from OPC UA information models are elicited. After that, a type instantiation algorithm is designed to support the generated code. Finally, a design for the source code generation tool is constructed. Functional prototypes are constructed for both the type instantiation algorithm and the source code generation tool.

The elicited requirements indicated that the type instantiation algorithm should be separated from the source code generation. The designed type instantiation algorithm creates instances of OPC UA types by reading the server address space on run-time. The designed source code generation tool generates Java classes that use the instances created by the algorithm.

The results of this thesis are used in the future development of the Prosys OPC UA Java SDK. The prototypes are developed further by implementing missing requirements and the elicited requirements are used for validating the final product.

Keywords: OPC UA, source code generation, information modeling, type instantiation

AALTO-YLIOPISTO
SÄHKOTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Eero Laukkanen

Työn nimi: Java-ohjelmakoodin generointi OPC UA -tietomalleista

Päivämäärä: 9.9.2013 Kieli: Englanti Sivumäärä:7+57

Automaatio- ja systeemitekniikan laitos

Professuuri: Automaation tietotekniikka Koodi: AS-116

Valvoja: TkT Ilkka Seilonen

Ohjaaja: DI Jouni Aro

OPC Unified Architecture on teollinen tiedonsiirtomäärittely, jonka eräs ominaisuus on tiedon mallintaminen. Tämä ominaisuus mahdollistaa siirrettävän tiedon mallintamisen oliomallilla, joka on samankaltainen kuin olio-ohjelmointikielissä. Tietomallin hyödyntäminen on kuitenkin haastavaa tämänhetkisellä Prosys OPC UA Java SDK:lla.

Tässä työssä tutkitaan mahdollisuuksia helpottaa tietomallien käyttöä. Ensin muodostetaan vaatimuksia lähdekoodin generoinnille OPC UA -tietomalleista. Tämän jälkeen suunnitellaan tyyppien instantiointialgoritmi tukemaan generoitavaa koodia. Lopuksi suunnitellaan lähdekoodin generointityökalu. Sekä instantiointialgoritmille että generointityökalulle tehdään toiminnalliset prototyypit.

Kerättyjen vaatimusten perusteella tyyppien instantiointialgoritmin tulee olla erillään lähdekoodin generoinnista. Suunniteltu instantiointialgoritmi luo instansseja OPC UA -tyypeistä lukemalla palvelimen osoiteavaruutta ajonaikana. Suunniteltu lähdekoodin generointityökalu generoi Java-luokkia, jotka käyttävät algoritmin luomia instansseja.

Työn tuloksia tullaan käyttämään Prosys OPC UA Java SDK:n jatkokehityksessä. Prototyyppejä kehitetään toteuttamalla puuttuvia vaatimuksia ja kerätyillä vaatimuksilla todennetaan lopullisen tuotteen toiminnallisuus.

Avainsanat: OPC UA, lähdekoodin generointi, tietomallinnus, tyyppien instantiointi

Preface

I would like to thank my instructor Jouni Aro for mentoring me through the world of OPC UA. We had many inspiring discussions about how things should be done and how much work we still have to do. Another thanks goes to Ilkka Seilonen who first gave me a glimpse of what the thesis should be about and later on became my supervisor. All the comments and improvements were appreciated. Thanks Ilkka and Jouni!

Siltamäki, 9.9.2013

Eero I. Laukkanen

Contents

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
COM	Component Object Model
DCOM	Distributed Component Object Model
DSL	Domain Specific Language
GUI	Graphical User Interface
IDE	Integrated Development Environment
LINQ	Language Integrated Queries
MVC	Model-View-Controller
OLE	Object Linking and Embedding
OPC	OLE for Process Control
SCADA	Supervisory Control And Data Acquisition
SDK	Software Development Kit
UA	Unified Architecture
XML	Extensible Markup Language

1 Introduction

1.1 Background

In industrial automation and control systems, devices and pieces of software from different manufacturers need to be able to communicate with each other. This ability is called interoperability. The Classic OPC specification enabled interoperability on communication protocol level by using the DCOM technology which was available on every Microsoft Windows PC. Nowadays, the interoperability is not a problem anymore on the protocol level, but instead the information model used by different manufacturers varies. This means that software components cannot be reused between different manufacturers. The usage of the successor of the Classic OPC, OPC Unified Architecture, can lead to interoperability on the information level too.

OPC Unified Architecture introduces a concept of information modeling. Compared to the Classic OPC, OPC UA enables usage of higher level semantics when organizing information on the server. Variables are encapsulated into objects that can be modeled with type hierarchies and inheritance, in a similar manner to object-oriented programming languages. The objects can have references to other objects, which allows modeling relations between objects. (OPC Foundation 2012b)

While the OPC UA specification has enough features for achieving interoperability on the information level, in practice all these features are not available yet. OPC Foundation provides communication stacks that implement the low-level communication protocol of the OPC UA, but the stacks do not provide functionality for using the information models. In some commercial OPC UA Software Development Kits (SDKs), the usage of information models is achieved with source code generation tools which exist for C# and C++ programming languages (Unified Automation 2013; CommServer 2013; OPC Foundation 2011b). These tools enable users to create information models with a graphic user interface (GUI) and then save the models as standard information model XML files (OPC Foundation 2012e). The tools generate C# or C++ classes that the server developers can use to create and use the objects of the information models.

1.2 Objectives and scope

Prosys is a software company specialized in using the Classic OPC and the OPC UA for products and services. Prosys OPC UA Java SDK in its current state supports importing custom information models to the server address space, but

actually creating and using the objects defined in the information models has to be done manually. The main objective of this thesis is to identify how the usage of information models could be made easier with the Prosys OPC UA Java SDK.

To reach the main objective, this thesis tries to answer the following research questions:

1. *What are the requirements for source code generation from OPC UA information models?* The requirements would be a starting point for answering the following questions.

2. *How should the generated source code be used in OPC UA applications?* The answer to this question defines the format of the generated source code and how the code integrates to other source code of the application.

3. *How should the source code generation be done in practice?* The answer to this question defines a design for the source code generation tool.

The actual implementation of a finished and tested source code generation tool is not in the scope of this thesis. However, the presented requirements and designs can be used as a base knowledge for future development.

Scope of this thesis does not include the creation of information models. Standard XML files representing information models can already be created with GUI tools (Unified Automation 2013; CommServer 2013; HB-Softsolution 2011). Currently newest version of XML Schema (OPC Foundation 2011a) is used as a basis for the source code generation.

1.3 Research methods

Previous knowledge about source code generation in the context of OPC UA is limited, because there does not exist published studies close to the subject. Therefore this thesis is conducted as exploratory research. The OPC UA specification is used as a source of information to ensure compliance with the specification. The existing source code generation tools are studied and previous studies close to the subject in the context of OPC UA and source code generation are examined.

Functional prototypes are constructed to examine practical limitations of the proposed designs. Prototyping process was kept informal, to allow tight interaction between the prototype and the design. Using prototypes validates that the designs are feasible for implementation, but further testing of the designs is needed to validate their usage in practice.

1.4 Structure of the work

The structure of this thesis is as follows. First, a literature study is performed, to gain background understanding of the subject (Section 2). Then, based on the gathered knowledge, use cases and requirements for using information models are determined (Section 3). Based on the elicited requirements, an extension to the Prosys OPC UA Java SDK is designed so that instances can be created from OPC UA types (Section 4) and Java code can be generated from standard XML files that represent OPC UA information models (Section 5). Finally, conclusions and future work are discussed (Section 6).

2 Background

In this section, subjects related to this thesis are studied and literature under those subjects is examined and referred shortly to give the reader a brief but satisfactory understanding. By reading this section, the reader should understand the concepts and terms that are used in the later sections.

First subject related to the thesis is inevitably the industrial communication specification OPC Unified Architecture. While there exist many introductions to the subject (Wolfgang Mahnke, Leitner, and Damm 2009; Palonen 2010; Hiltunen 2012), another one is provided here in the context of this thesis. Second studied subject is source code generation in general and in the context of OPC UA. Based on previous implementations where code generation is done (Unified Automation 2013; OPC Foundation 2011b), different requirements of source code generation are examined. Research by Goldschmidt and W. Mahnke (2012) is also introduced. They studied use cases for domain specific languages in the context of OPC UA.

2.1 Introduction to OPC Unified Architecture

OPC Unified Architecture is an industrial communication specification meant to replace the Classic OPC specifications which enabled users to read, write and monitor data, transmit alarms and events and access historical data on a remote computer. The motivation for the original specification was to provide interoperability between devices and software from different manufacturers. While the functionality of the Classic OPC specifications might have been sufficient, the dependency on COM and DCOM technologies severely limits the communication possibilities between networked computers nowadays. OPC UA solves the networking issues by using open standardized protocols. (Wolfgang Mahnke, Leitner, and Damm 2009, p. 3–9)

The latest OPC UA specification was released by OPC Foundation in 2012 and it is divided in 13 parts (Table 1). Compared to the Classic OPC, OPC UA contains multiple improvements:

- communication is based on open protocols, so there is no dependency on COM and DCOM technologies (OPC Foundation 2012e)
- security model (OPC Foundation 2013e), that ensures that information can be transferred over Internet safely by encrypting data and provides authentication and authorization

– information model (OPC Foundation 2012b), that enables more sophisticated modeling of the data.

In this thesis, the focus is set on the information model and its usage.

Table 1: The parts of the OPC Unified Architecture specification. (OPC Foundation 2012a)

Part	Description
Part 1 – Overview and Concepts	Self-explanatory.
Part 2 – Security Model	Defines how secure connection is ensured.
Part 3 – Address Space Model	Defines the underlying meta model that is used in constructing the information model.
Part 4 – Services	Defines service interfaces between clients and servers.
Part 5 – Information Model	Defines the default information model of OPC UA servers.
Part 6 – Mappings	Defines how parts 2, 4 and 5 are implemented using physical network protocols.
Part 7 – Profiles	Defines profiles that are subsets of services that certain kind of, e.g. embedded, servers have to provide.
Part 8 – Data Access	Defines how clients can read, write and monitor data values on servers.
Part 9 – Alarms and Conditions	Defines how servers can send alarms and conditions to clients.
Part 10 – Programs	Defines how programs can be modeled with OPC UA.
Part 11 – Historical Access	Defines how clients can access historical values on servers.
Part 12 – Discovery	Defines how clients can discover servers automatically. At the time of writing, this part was not released yet.
Part 13 – Aggregates	Defines how clients can request derived values from raw historical or buffered real time data.

OPC UA communication adapts the client-server architecture, meaning that there are always two identities communicating to each other and the one that starts the communication is the client. There can be multiple clients communicating with one server, but those clients cannot communicate with each other directly. An OPC

UA client communicates with the server by sending requests to which the server answers by sending responses. The format of the requests and the responses are defined as OPC UA services (OPC Foundation 2012c).

In Classic OPC, data values on a remote computer could be described only with a tag name and some rudimentary information like the engineering unit (Wolfgang Mahnke, Leitner, and Damm 2009, p. 19). The tags could be also put into folders and hierarchies could be constructed with the folders. In contrast, OPC UA introduces information modeling concept similar to object-oriented programming languages with which it is possible to define types and instantiate objects based on those types. Types can be organized into hierarchies where subtypes inherit the properties of supertypes (OPC Foundation 2012b). This allows clients to process data based on a type, not only on a tag name. Next, the OPC UA information modeling capabilities are described more deeply. After that, the OPC UA services are presented.

2.2 OPC UA object model

OPC UA information models are constructed with two metamodels with different abstraction levels. The high-level metamodel is the *object model* which is used to semantically model the address space, similarly to the type systems in object-oriented programming languages. The object model is constructed with the low-level metamodel *node model* which is an implementation model of the address space. It is not a general modeling structure but instead designed for modeling the object model. (OPC Foundation 2012b)

OPC UA object model (Figure 1) has the following properties:

- Objects encapsulate variables, methods and other objects.
- Objects can have references to other objects.
- Objects can be instantiated from object types that define the structure of their instances.

Objects can have two kinds of variables:

Data variables are variables that represent the values of their parent and can have child variables

Properties are variables that describe the characteristics of their parent and cannot have child variables

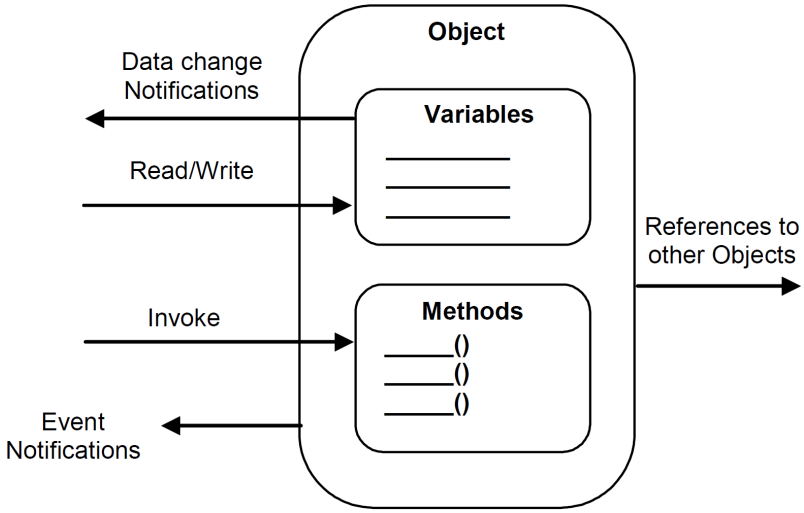

Figure 1: OPC UA object model. (OPC Foundation 2012b)

Data variables behave like objects, in the sense that they can be instantiated from variable types that define their structure. Differing from objects, data variables are not allowed to have methods or child objects and they always require an object that encapsulates them.

Objects can be instantiated from object types, in which case an instance of the object type is added to the server address space and it will have the same structure the object type has. In addition, object types can be subtypes of other object types, in which case also the variables and methods of the supertypes are added to the instance. This is called inheritance; a subtype inherits the structure of its supertype. With rich type hierarchies, UA clients can process objects on the server based on not only the types of instances but also the supertypes of instances. Subtypes can also override the components defined in the supertype, but then the component that overrides has to have the same type or subtype as the component that is overridden. (OPC Foundation 2012b)

Type information is always available for OPC UA clients to read from the server address space. This way, the clients can understand the structure of the types without sharing knowledge outside the OPC UA protocol.

OPC Foundation has specified a base information model (OPC Foundation 2012d) that all OPC UA servers should provide. Building on top of that, different industries can model their common domain information models. Software can be built to understand the common information model, achieving interoperability at the information level. If needed, the common information model can also be extended, which does not break the interoperability.

OPC Foundation maintains companion specifications for information models for devices (OPC Foundation 2013c), analyser devices (OPC Foundation 2013b), programmable logic controllers (OPC Foundation and PLCopen 2010) and the object model of the ISA-95 specification (OPC Foundation 2013d). Other parties have also implemented information models for, e.g., building automation systems (Granzer and Kastner 2012) and smart grids (Lehnhoff et al. 2012).

While the data on UA servers is represented as objects, UA services are used directly with the underlying data model, the node model of the server address space. For example, when reading and writing variables of an object, the data access service is used with a node identifier of the variables, bypassing the object model (OPC Foundation 2012c).

2.2.1 Node model

OPC UA node model (Figure 2) builds up the server address space. The address space consists of nodes which have attributes and references to other nodes. Each node has a node class attribute which represents an element of the object model. There are seven different node classes: objects, variables, object types, variable types, data types, reference types and views. Others are self-explanatory, except data types which represent types of the values of variables and views which represent subsets of the address space.

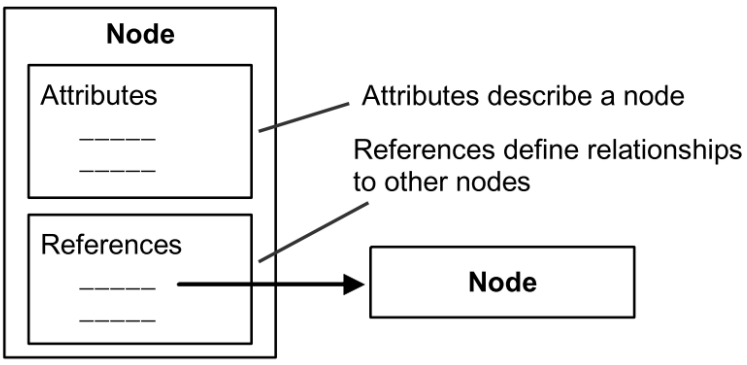

Figure 2: The node model. (OPC Foundation 2012b)

In addition to node class, there are two other common node attributes that should be introduced:

Node identifier uniquely identifies each node. It consists of a namespace and an identifier.

Browse name uniquely identifies each node in the context of a parent node in a type definition. It consists of a namespace and a name.

A namespace is a Uniform Resource Identifier (URI) that identifies the naming authority. Namespaces are needed to make node identifiers and browse names unique among different information models. Each information model should have a unique namespace. This way, the maintainers of the information models need to care about naming conventions only inside their own model. In service requests, the namespaces are referred by the namespace index that corresponds to their position in the namespace array of the server. The namespace of the base information model is http://opcfoundation.org/UA/ and its namespace index is always 0.

Node classes have different attributes according to the class. Other standard attributes are introduced when needed in this thesis. Users of the OPC UA are not allowed to extend the node attributes or create their own node classes.

References of a node point from the source node to the target node and have a reference type (Figure 3). Source and target nodes are identified by their node identifiers. Users of the OPC UA are allowed to create their own reference types. The OPC UA specification categorizes all reference types under specific two:

Hierarchical references are used to form hierarchies of nodes. Their only limitation is that a node cannot have a hierarchical reference to itself. Thus, the formed hierarchies are allowed to have loops in them. A property variable cannot have structure, so it cannot be the source node of a reference of this type.

Non-hierarchical references should not be presented as spanning hierarchies. They can be used rather freely, because there are no restrictions how to use them.

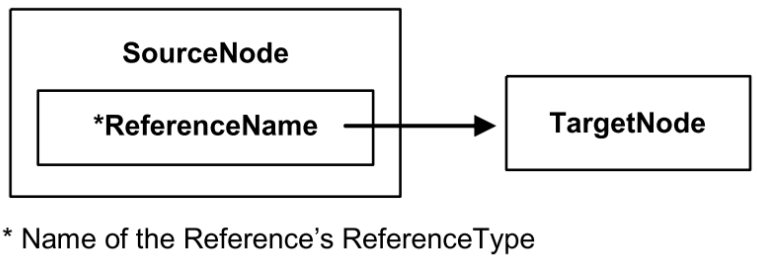

Figure 3: The reference model. (OPC Foundation 2012b)

The concrete reference types that are used to model the objects in the address space are subtypes of either hierarchical reference type or non-hierarchical reference type (Figure 4). In the scope of this thesis the following reference types are introduced:

HasChild is an abstract hierarchical reference type. It adds the restriction that no loops are allowed in the hierarchy that is formed with the reference type.

HasSubtype is a concrete has-child reference type. Object and variable type hierarchies are formed with this reference type.

HasComponent is a concrete has-child reference type. Parent objects and data variables have this reference to their child objects, variables or methods.

HasProperty is a concrete has-child reference type. Parent objects and data variables have this reference to their properties.

HasModellingRule is a concrete non-hierarchical reference type. Every node that is instantiated when a type is instantiated has a reference of this type to a modeling rule object. Modeling rules define how those nodes are managed during instantiation.

HasTypeDefinition is a concrete non-hierarchical reference type. Every instance of a type has a reference of this type to its type definition node.

The OPC UA specification defines a standard graphical notation for visualizing nodes and their references (Figure 5). Each node class has an own symbol. References are either displayed as simple arrows with the name of the reference type on it or as special arrows for certain reference types. The browse name of the node is shown on the symbol.

The OPC UA object model can be represented with the node model. The following rules must hold:

- objects, variables, methods, object types and variable types are represented as single nodes with the respective node classes
- object and variable nodes have a HasTypeDefinition-reference to their type
- objects have HasComponent-references to their child objects, data variables and methods
- objects have HasProperty-references to their properties
- types have HasSubtype-references to their subtypes.

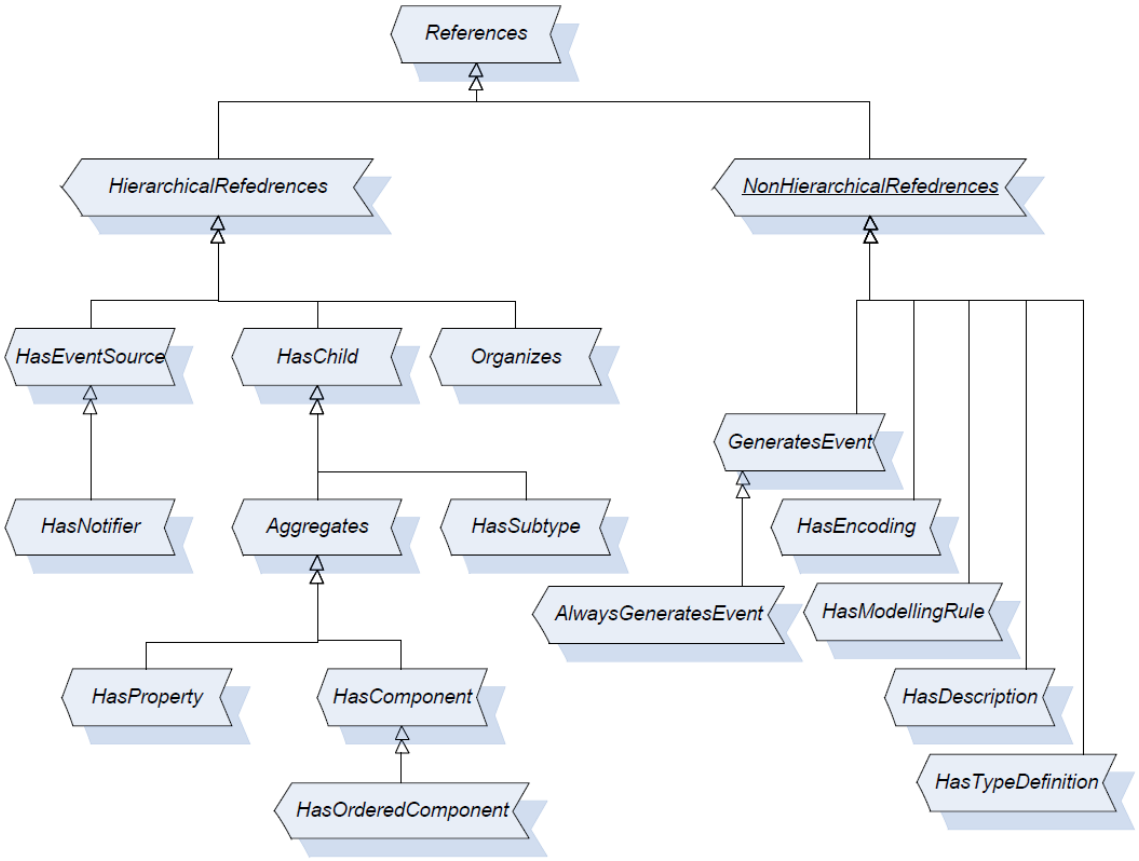

Figure 4: Standard reference type hierarchy. (OPC Foundation 2012b)

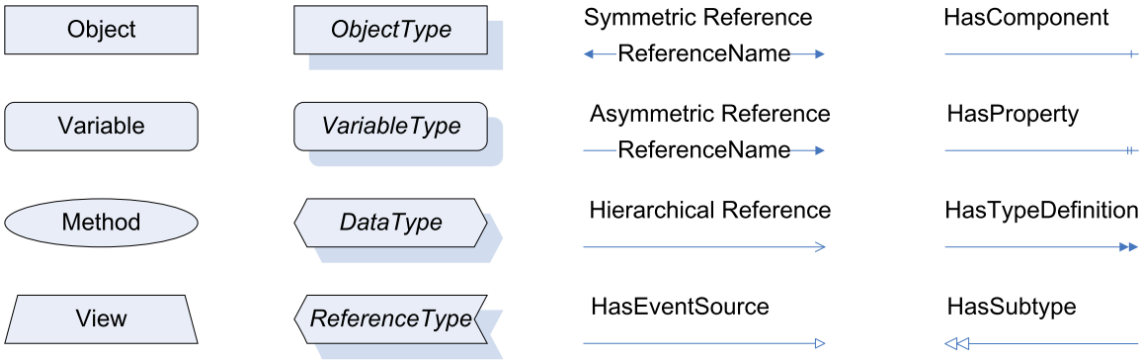

Figure 5: Simple notation of nodes and references. The attributes of nodes are not visible. (OPC Foundation 2012b)

2.3 Instantiation in OPC UA

So far in this section, it has been covered that in the address space of an OPC UA server there can be types and instances of those types (Figure 6). However, it has not yet been defined what does it actually mean that an instance has a type definition.

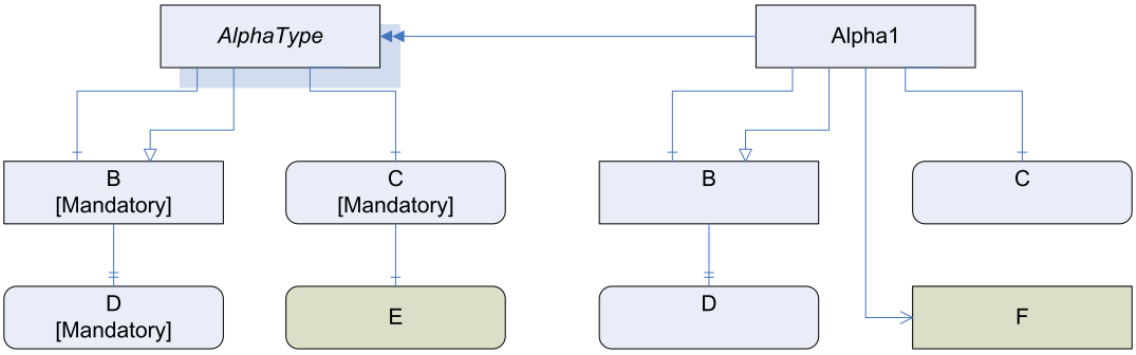

Figure 6: A type definition AlphaType and its instance Alpha1. Modeling rules are shown in the figure as text below the browse names of the nodes. The E-node has no modeling rule. (OPC Foundation 2012b)

The type instantiation process is split to three separate steps in the OPC UA specification (OPC Foundation 2012b):

1. Creating an instance declaration hierarchy
2. Merging instance declaration hierarchies
3. Creating instances and references

The steps are next introduced separately.

2.3.1 Creating an instance declaration hierarchy

An instance declaration hierarchy consists of the type definition node and its instance declarations. An instance declaration is defined in the specification as:

> "An InstanceDeclaration is an Object, Variable or Method that references a ModellingRule with a HasModellingRule Reference and is the TargetNode of a hierarchical Reference from a TypeDefinitionNode or another InstanceDeclaration." (OPC Foundation 2012b)

For example, in the Figure 6 the B-, the C-, and the D-node are instance declarations, because they are connected through hierarchical references from the type definition node and they have a modeling rule. The E-node is not an instance declaration, because it does not have a modeling rule. Modeling rules specify the purpose of the instance declarations. They are described thoroughly in Section 2.3.3.

Each instance declaration has a unique *browse path* that is constructed by following the hierarchical references from the type definition node and collecting the browse names of the nodes. For example, the node D has a browse path "/B/D"

in the Figure 6 (Table 2). In the textual form of the browse path, the namespaces of the browse names are often omitted. An instance declaration can have multiple browse paths, because there can be loops in the hierarchy formed by hierarchical references.

Table 2: Browse paths of the nodes in the instance declaration hierarchy of the AlphaType in Figure 6.

Node	Browse path
AlphaType	/
B	/B
C	/C
D	/B/D

Before the instance declaration hierarchies have been merged, it is important to handle the instance declarations based on their browse paths. Thus, when the instance declaration hierarchy is constructed, references to other instance declarations are handled as reference to browse paths instead of node identifiers.

2.3.2 Merging instance declaration hierarchies

Instances of a type inherit the structure of their supertypes too. Therefore instance declaration hierarchies need to be merged to collect all the needed information for instantiation. A fully inherited instance declaration hierarchy is formed when the instance declaration hierarchy of a type is merged with the fully inherited instance declaration hierarchy of its supertype. If a type has no supertype, then no merging needs to be done.

Simplified rules for merging are the following:

- An instance declaration is added from the supertype to the subtype, unless there is already an instance declaration with the same browse path. All references of the instance declaration are added as well.
- If there exists an instance declaration with the same browse path in the subtype, only the references of the instance declaration are merged from the supertype to the subtype.
- A reference is added from the supertype to the subtype, unless there is already a reference with the same source and target paths and with the same type or subtype of it.

When a similar instance declaration or reference exists in the instance declaration hierarchy of the supertype then the subtype overrides that instance declaration or reference. There are quite a many specific rules for overriding that consider modeling rules, attributes of the nodes and type definitions of the nodes. General idea of those rules is that the subtypes cannot contradict the supertypes and can only make the requirements for the instances more specific, not less specific. These rules should be checked when the model is designed and before the instantiation is done.

2.3.3 Creating instances and references based on modeling rules

The fully inherited instance declaration hierarchy contains all the information that is needed for creating instances of a type. The modeling rules of instance declarations decide how the instance declarations are instantiated. Each modeling rule has a special semantic meaning on their own, but they also have a property called *naming rule*. This rule can be one of the three:

Mandatory means that a similar node with the same browse path as the instance declaration shall be found from the instance.

Optional means that the instance may or may not have a similar node with the same browse path as the instance declaration.

Constraint is used for modeling rules for instance declarations that typically are not found in the instance. Instead, the instance declarations define other kind of semantics for the instance declaration.

The OPC UA specification defines five modeling rules:

Mandatory has mandatory naming rule and fulfills the specification for that naming rule above.

Optional has optional naming rule and fulfills the specification for that above.

ExposesItsArray has constraint naming rule. It is used to model that a variable type which has an array of values should expose each of those values as a node in the address space. All the nodes should be similar to the instance declaration marked with this modeling rule.

OptionalPlaceholder has constraint naming rule. It is similar to the Optional modeling rule, but is used when the browse name of the instantiated node is not known in the type definition. There can be multiple instantiated nodes similar to the instance declaration marked with this modeling rule.

MandatoryPlaceholder has constraint naming rule. It is similar to the OptionalPlaceholder modeling rule but means that there should exist at least one similar node to the instance declaration marked with this modeling rule.

Users of the OPC UA are allowed to create their own modeling rules.

If only the standard modeling rules are used, then only the mandatory instance declarations need to be instantiated. The OPC UA specification leaves the OPC UA server much to decide about the instantiation:

> "The Nodes within the newly created hierarchy may be copies of the InstanceDeclarations, the InstanceDeclaration itself or another Node in the AddressSpace that has the same TypeDefinitionNode and BrowseName." (OPC Foundation 2012b)

The specification does not actually define that new nodes are created for the instances. It just requires that the instances need to have the same structure as the type definition. Similar to the overriding rules discussed in Section 2.3.2, the nodes in the instances can have type definitions that are subtypes of the types of the instance declarations.

Other things that the server must decide on instantiation are:

- Instance declaration with multiple browse paths can be represented with either multiple nodes or a single node.
- Non-hierarchical references defined in the instance declarations can be either present or not. However, HasTypeDefinition-references are required.

2.4 OPC UA Services

OPC UA clients get access to data on servers through OPC UA services (OPC Foundation 2012c) which are interface definitions between a client and a server. Understanding the services is important in the context of this thesis, because the meaning of source code generation is to create a mapping between objects and service interfaces. On the server side, code generation could create a mapping between service requests and objects that map the request to the underlying data. On the client side, code generation could create objects with which service requests could be made.

OPC UA services are defined as abstract services, meaning that the actual implementation details have been left out. Because of this, OPC UA services can be

used with two different implemented protocols, Binary TCP and XML Web Services. One could implement his own protocol for UA services, but usually those provided by the specification are sufficient. (OPC Foundation 2012c)

Typical service usage scenario is when a client sends a request to a server and after that the server sends a response back to the client. Exceptional situations occur, if the network connection is broken or the request is invalid in some way. These situations are handled with timeouts and status codes in service responses. (OPC Foundation 2012c)

OPC UA services are organized into nine service sets (Table 3). Out of these, Discovery, Secure channel and Session service sets are related to the underlying connection and the rest are used to view, modify and use the data that is available from the information model. In the context of this thesis, the latter ones are more interesting:

Node management service set is meant for adding and deleting nodes and references to and from the server address space.

View service set allows clients to browse and query the address space. By browsing, nodes connected to a certain node can be requested. By querying, nodes with a certain type can be requested.

Attribute service set contains services for reading and writing attributes of nodes. Also historical values can be read and updated with this service set.

Method service set contains Call-service with which methods can be called.

Monitored item and subscription service sets are used to subscribe for notifications from the server based on attribute value changes or events. Pushing data from a server to a client is done by long polling, meaning that the client sends a request to the server and the server sends a response back only after it has a notification to send. (OPC Foundation 2012c)

2.5 Source code generation

Source code generation means creating source code automatically based on some initial data model. By automatically generating source code, manual typing of the code with similar structure is avoided. Source code generation has multiple use cases such as

Table 3: The OPC UA service sets. (OPC Foundation 2012c)

Service set	Use case
Discovery	Discover servers and their security settings.
Secure channel	Services related to the security model.
Session	Maintain the session between a client and a server.
Node management	Modify the address space.
View	Browse through the address space.
Attribute	Read and write attributes of nodes.
Method	Call methods.
Monitored item	Setup monitoring for attribute value changes or events.
Subscription	Subscribe for attribute value changes or events.

Reuse: the programming language does not support encapsulating the structure into a reusable component (Völter 2003)

DSLs: the program is defined with a domain specific language (DSL) to give a higher abstraction level syntax for domain specific requirements (Völter 2003)

Translation: the data model is defined in some other format than the programming language (Sheard 2001)

Performance: instead of directly writing unclear efficient code, the source code is generated from a written specification (Sheard 2001)

2.5.1 Patterns

Völter (2003) describes multiple patterns how to do source code generation. The most relevant to this thesis are

Templates + filtering where the source model is first filtered and the data gained from those filters is applied to a template (Figure 7). An example of filtering would be XSLT (W3C 1999) where XML file is read, filtered for data and another XML file or text file is formed. Users of the generator can modify the templates and the filters for their needs. The templates resemble the result that is tried to achieve, but in place of the actual values fetched from the source model the templates have variables or control code to get the data.

Templates + metamodel which is similar to the templates + filtering -pattern (Figure 8). Instead of just filtering the source model, a metamodel is produced from it and the data of this metamodel is applied to the templates.

API-based generators which do not use templates to produce source code, but instead form an abstract syntax tree (AST) of the source code (Figure 9). The AST can be either built by a compiler or unparsed to source code. The user of the generator calls functions on an API that has a higher abstraction level than the AST.

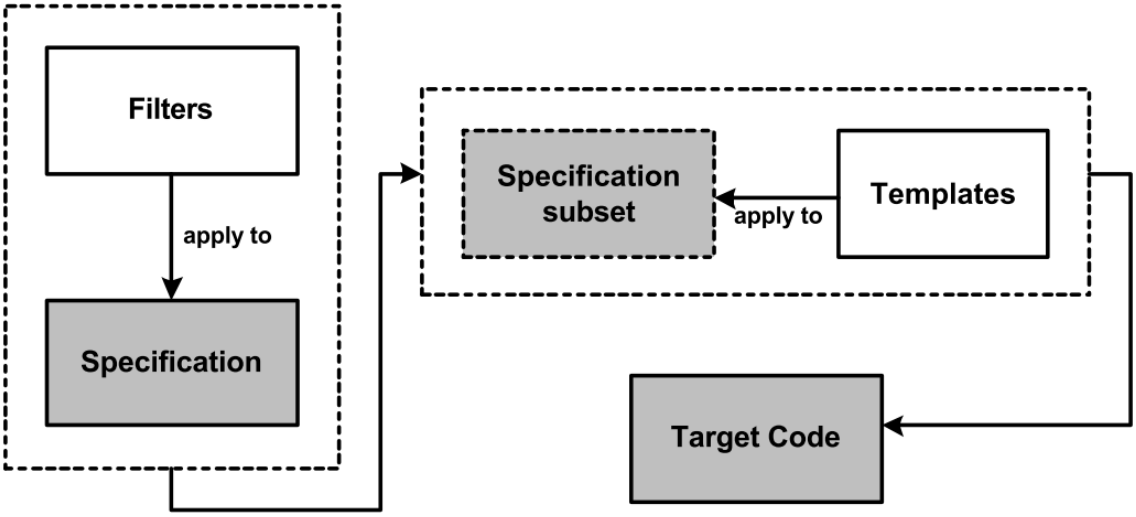

Figure 7: Templates + filtering pattern. (Völter 2003)

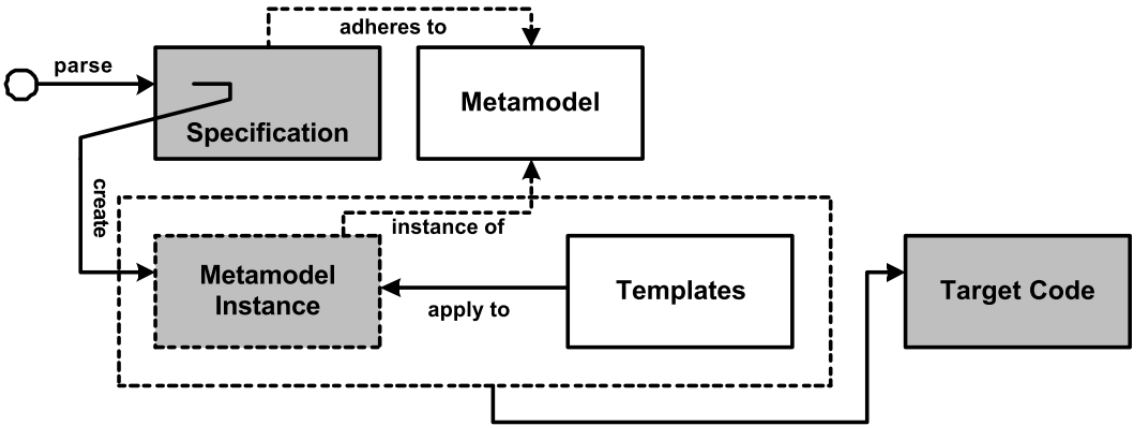

Figure 8: Templates + metamodel pattern. (Völter 2003)

Other patterns that Völter describes are not useful in the context of this thesis, because those patterns generate source code from other source code. In this thesis the only input for the generation is an OPC UA information model.

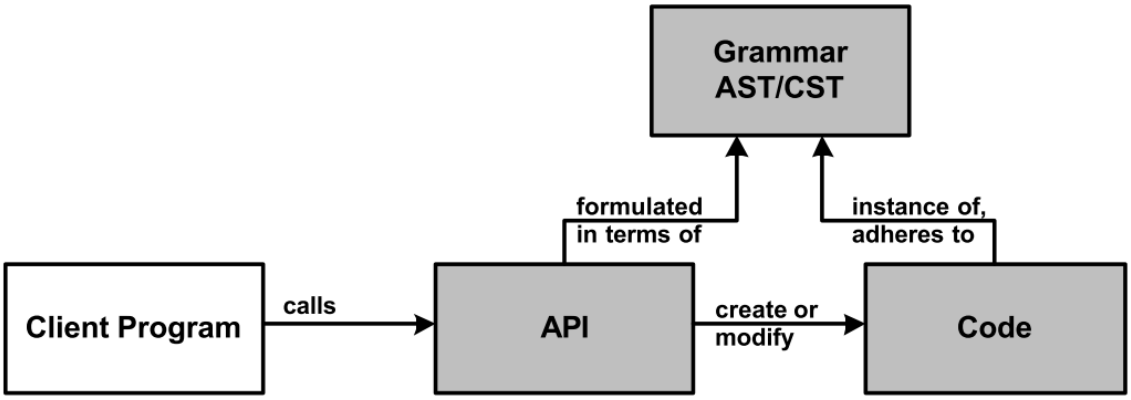

Figure 9: API-based generators pattern. (Völter 2003)

When using templates to generate source code, the next question is which template language is used. It is often required to have somewhat restricted template language compared to Turing-complete programming languages (Arnoldus 2011; Parr 2004). A restricted template language enforces separation of the template and the model and makes the templates inherently cleaner and easier to read. Some template languages, such as Mustache (Wanstrath 2013), call themselves ”logic-less” to illustrate their restrictive nature. When logic inside the templates is limited, it is usual that data needs to be preprocessed to fit into the templates. Thus, it is common to use the templates + metamodel pattern with logic-less templates.

The generated source code is often used in conjunction with handwritten source code. For example, code generation can be used to implement only a specific part of a larger program and other parts are implemented by hand. The integration of the generated and handwritten source code is not straightforward, because for example if the generated code is modified to achieve the integration, then regenerating the code would erase those modifications. Thus, it is required that the integration is not achieved by modifying the generated code, but instead by using a design that makes the modifications unnecessary.

Völter (2003) describes multiple methods for integrating the generated code to the non-generated code (Figure 10):

a) the generated code can call non-generated code in libraries

b) the opposite of a): non-generated code can call generated code

c) similar to b), but the generated code implements an interface which the non-generated code uses

d) generated classes can be subclasses of non-generated classes

e) same as d), but the generated classes implement abstract methods defined in non-generated classes

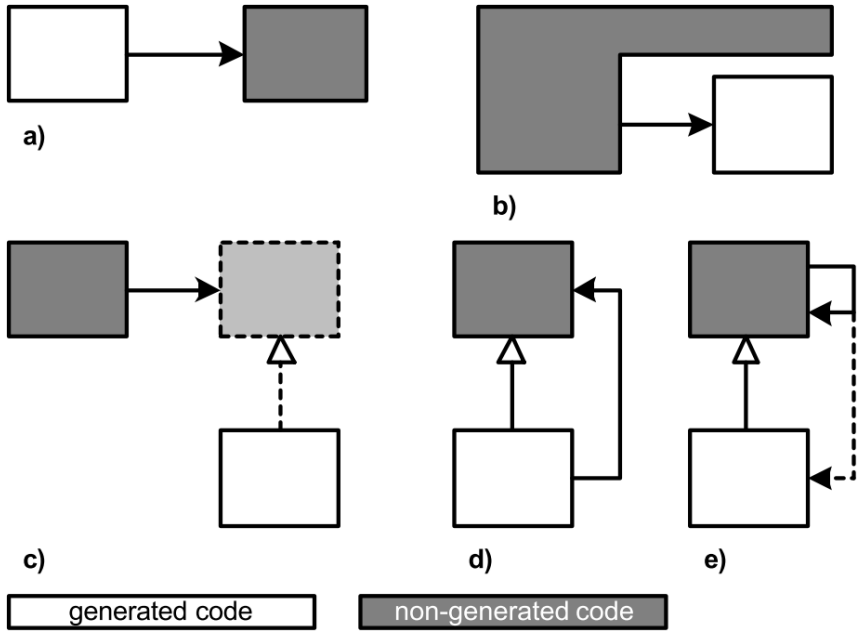

Figure 10: Ways to integrate the generated code to the non-generated code. (Völter 2003)

2.6 Source code generation tools for OPC UA

In the context of OPC UA, source code generation is mainly used for translating the OPC UA information models to be used with target programming languages. Also in the OPC UA Java Communication Stack (OPC Foundation 2013a), some parts of the OPC UA specification are used for source code generation.

The benefit of generating source code from the information models is that the developers of OPC UA applications can leverage the type information present in the models in their applications. With the help of integrated development environments (IDE), developers can use auto-completion to quickly type code that would otherwise be prone to typing errors. The code will not compile if a developer tries to use a variable that does not exist in the information model. The source code generation needs to be static, meaning that it has to be done before compilation and results in static source code files (Sheard 2001).

There exist four commonly known tools related to OPC UA information modeling and source code generation (Table 4). With the help of tools, OPC UA information models can be created with a graphical user interface (GUI) and then code can be

generated based on the model. The format of the code depends on the generator and the SDK for which the code is generated for. Models can be defined either in ModelDesign-format which is used for the C# code generator called ModelCompiler or in UaNodeSet-format which is defined in the OPC UA specification (OPC Foundation 2012e) and is a generic way to define OPC UA address spaces. There exists also an older NodeSet-format which is similar to the UaNodeSet-format but is not part of the OPC UA specification. All mentioned model formats are defined in Extensible Markup Language (XML), which enables saving the models as text files.

Table 4: Previous implementations related to source code generation.

Name	Version	Modeling GUI	Model format	Code generation	Language
ModelCompiler	1.01	–	ModelDesign	✓	C#
UaModeler	1.2.0	✓	UaNodeSet	✓	C, C#, C++
CAS	3.0.2	✓	ModelDesign	✓	C#
Comet	0.2	✓	NodeSet	–	–

OPC Foundation ships a code generating tool with OPC UA SDK (OPC Foundation 2011b) called ModelCompiler. With the tool, C# source code can be generated from information models defined in ModelDesign-format. OPC Foundation has stated that ModelCompiler will become obsolete and is replaced with a tool that uses the UaNodeSet-format (Armstrong 2013). Unified Automation has made an OPC UA modeling tool called UaModeler (Unified Automation 2013). With it, UA information models can be created with a GUI, the models can be saved in UaNodeSet-format and C, C# and C++ source code can be generated from the model. CAS Model Designer (CommServer 2013) by CAS has also a GUI for creating OPC UA information models. CAS Model Designer uses the ModelCompiler by OPC Foundation to generate C# code. Comet UA Model Designer (HB-Softsolution 2011) has a GUI for creating OPC UA information models. The models can be saved in the NodeSet-format, but source code cannot be generated with the Comet UA Model Designer.

There are no standard guidelines which would specify what kind of code should be generated from the OPC UA information model. Some guidance can be imitated from the previous implementations. Since ModelCompiler is becoming obsolete, only UaModeler is studied further to understand use cases for source code generation. Current version of UaModeler, 1.2.0, can generate code for OPC UA servers

implemented in C or C++ and OPC UA clients implemented in C#.

UaModeler generates C++ server code for each defined object and variable type in the information model. Each type will generate a base class that contains all the generated code and a subclass which can be extended safely without losing changes if the code is regenerated. This is similar to the method d) described by Völter (2003) (Figure 10). Base classes contain code for accessing the variables and methods of the types. Using the generated code, applications can create new objects based on the types and update cached values for variables which are then transmitted to clients when requested. UaModeler also generates a node manager class which will manage all nodes which have the namespace of the newly created information model. When the OPC UA server is started, the node manager creates the type nodes of the new types and all the instance objects that are defined in the information model.

UaModeler is currently the only code generator for OPC UA clients. In the time of writing, the functionality of the generated code is limited; only constant information, like node identifiers, browse names and namespaces, is generated as multiple static classes. Nonetheless, this information is useful for browsing the address space and makes client development less error-prone.

Goldschmidt and W. Mahnke (2012) have studied what kind of domain specific language (DSL) support should be implemented for OPC UA. Code generation can be seen as a way to create a DSL that is used within a host programming language. OPC UA information model represents a domain model and that model can be leveraged via source code generation. Goldschmidt and Mahnke suggest multiple DSLs, but the most relevant for source code generation are:

Server: Schema mapping DSL would help OPC UA server developers to map the requests by clients to the data that is requested.

Client: Browse paths DSL would provide information about the structure of types on the server. With this information, clients could request values of components of objects.

Client: Calling methods & Properties DSL would provide object interfaces that would contain the types and names of the methods and properties. This would enable compile-time type and name checking for properties and methods.

In addition, Goldschmidt and Mahnke introduce a DSL prototype which uses generated code to make language integrated queries (LINQ). The prototype makes it easy to construct OPC UA queries and event filters, which is otherwise considered rather complex.

3 Requirements

In this section, current state of the Prosys OPC UA Java SDK and software requirements for improving the usage of information model capabilities are presented. It is intended that relevant requirements are first presented comprehensively, and afterwards it is decided which of the requirements are implemented in the scope of this thesis.

3.1 Prosys OPC UA Java SDK

Prosys provides an OPC UA software development kit (SDK) for developing OPC UA applications in Java (Prosys 2013). The SDK works on top of the Java stack (OPC Foundation 2013a) maintained by OPC Foundation. The Java stack provides a highly tested, low-level API for OPC UA communication, but the SDK makes the development of OPC UA applications effortless. Both client and server development can be done with the SDK.

Previously, two master theses have been done to develop the SDK further. Palonen (2010) implemented a way to create the address space of an OPC UA server from an XML-file. Hiltunen (2012) created an OPC UA client which has a graphical user interface so that OPC UA servers can be browsed easily.

Palonen (2010) studied how the basic support for information models could be added to OPC UA servers. He implemented a way to loading the server address space from an XML-file in ModelDesign-format and showed how data could be bound to the information model. He wrote his thesis in 2009 and in the current state of SDK, loading XML-files in UaNodeSet-format is already possible. Thus, in this thesis, it can be taken for granted that the server address space can be loaded from XML-files, and the focus is set to develop other aspects of the SDK further.

The types of the standard information model, the companion specifications and presumably the future standardized models are released in the UaNodeSet-format. Also custom information models can be created with UaModeler (Unified Automation 2013) and exported as UaNodeSet files. The SDK can load these information models on startup, and after that UA clients can browse the address space and see what types does the server support and what structure those types have.

To use the types defined in information models, the SDK contains some hand-written Java classes that represent standard object types. They map variable values to the address space and implement certain standardized features, such as alarms and conditions. The classes also instantiate the object types, by creating the child

instance nodes individually.

In the next sections, requirements for improvements to the SDK are discussed. First, improvements for type instantiation are considered. Then, source code generation for server-side, client-side and custom data types are discussed. Finally, the scope of this thesis is decided.

3.2 Type instantiation

While loading of type information is a critical feature, the SDK does not help the developers to instantiate those types to the server address space (Figure 11). Instead, the types need to be instantiated by hand now, which is error-prone and requires rework when the types change.

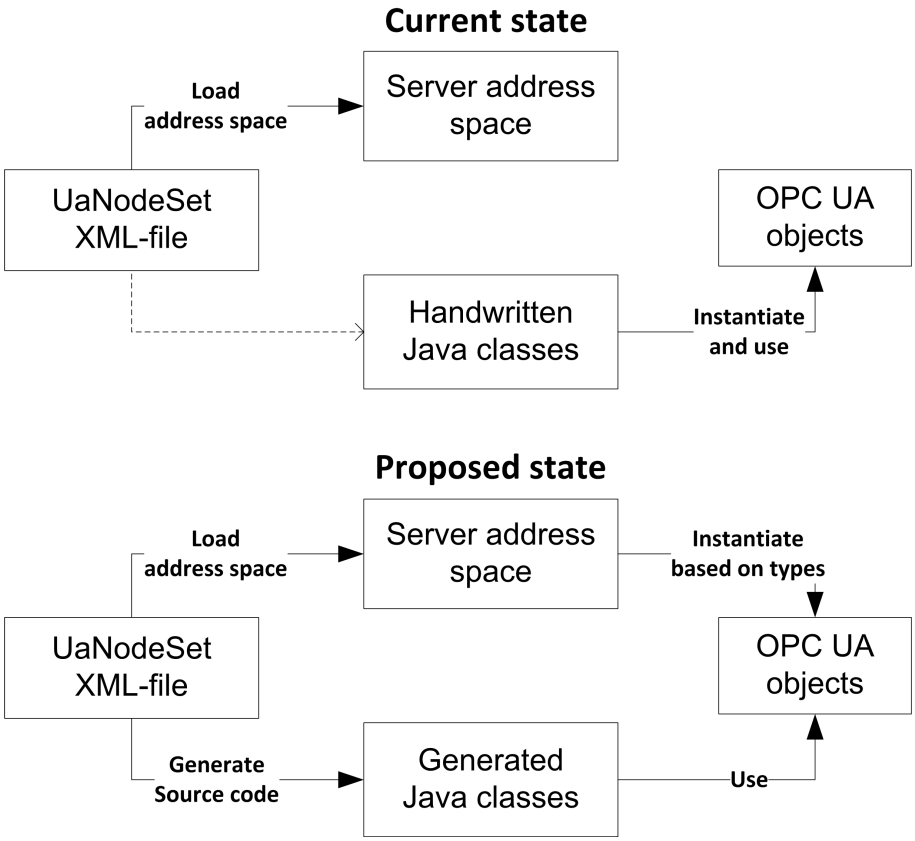

Figure 11: Overview of the requirements on the server-side. The current state and the state proposed by the requirements are illustrated.

UaModeler generates C++-code that handles type instantiation. However, this approach has some limitations compared to doing the instantiation at run-time:

- When the types change, new code has to be generated and the server application has to be rebuilt and restarted. Otherwise the instances reflect the type

declarations at generation-time, not at run-time.

- Type instantiation algorithm is already rather complex and it becomes even more complex when it is executed by the code generator. It is difficult to verify that the algorithm works correctly.

Because of these reasons, type instantiation should happen at run-time. This is possible by implementing an algorithm that reads the type address space.

The OPC UA specification does not specify whether new nodes are created during instantiation or not. However, in most cases new copies of the instance declarations are created, except for methods. Methods can be shared between instances, because they do not contain any state. To support reusing old nodes, the instantiation algorithm can be extended later on.

3.3 Source code generation

There are some common requirements for source code generation. First, like in the code generation implementations introduced in the background section, it should be possible to regenerate the code without losing any added code. Second, the generated code should not handle any special cases. Instead, they should be handled by adding custom code on top of the generated one. Third, source code generation should be only used when the features of the programming language are not sufficient for the task. Otherwise, maintenance of the code becomes unnecessarily dependent on the generation process.

An exception to the third requirement is when source code generation is used for performance optimization. While that can be the case for embedded OPC UA servers, the main focus in this thesis is set to source code generation for other purposes.

It should be possible to integrate source code generation to the build process of an application. This way, any changes in the information model that require modifying the application code have a potential to cause a compile error which forces the developer to fix the application code.

3.3.1 Namespace handling

The OPC UA node identifiers and browse names are made globally unique by their namespaces. Namespaces are URIs, but OPC UA servers refer to them internally by their namespace index, their position in the namespace array. This is done for efficiency.

In general, generated source code should use URIs when referring to namespaces, because the namespace indexes might not be known at the time the code is generated. For example, if source code for the SDK is generated, then the code is used by OPC UA servers which use different information models. Thus, their namespace arrays are not equal and the namespace indexes cannot be known when the code is generated. On the contrary, if code is generated just for a single application, then the namespace array can remain constant and the namespace indexes can be fixed during the generation.

In this thesis, source code for the SDK is generated, and therefore namespaces are handled by their URIs. This way, the generated code is reusable between different OPC UA applications.

3.3.2 Object and variable types on the server-side

Source code generation could produce Java classes that represent the object and variable types defined in OPC UA information models. OPC UA server developers could use those classes to map data to the nodes of the address space. Instead of mapping data by node identifiers or browse names, developers could leverage the Java type system to define the data source for each variable on a certain object. The Java classes could also be used to implement the methods of the objects and send event notifications from the object.

The handwritten object types in the current SDK work well, but are hard to maintain. They should be replaced with generated code and in addition, code could be generated for the types in the companion specifications.

Currently data is mapped in the SDK depending on the used NodeManager- and IOManager-components. One way to map data is to cache it: values are first written to the node objects and when the client needs them, they are read from the objects. Caching can be used in a current NodeManager implementation called NodeManagerUaNode which stores every node of the address space as an object in the memory of the server application.

Caching can be also seen as push based data mapping. Some part of the application has to push values first to the cache before they are available to the client. Pulling is another way to map data, meaning that the server actually pulls the data from somewhere else, when it needs it (Figure 12).

Whether to push or pull is a decision that the application developer should decide, because it depends on the data source and its properties. It is necessary to know whether the values can be cached and what time and performance costs there

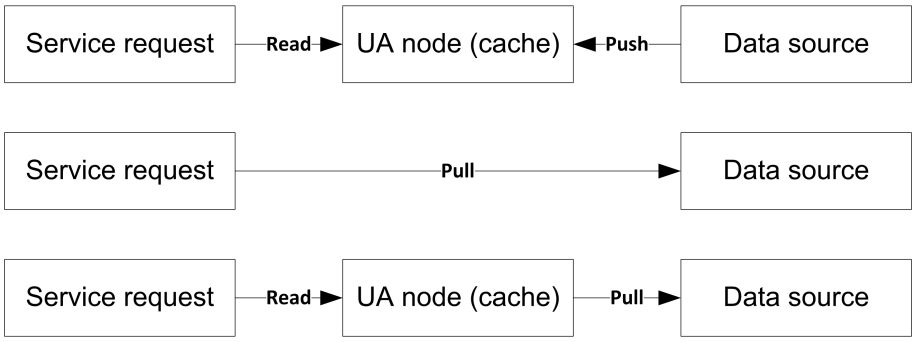

Figure 12: Different ways to map data to service requests.

are for requesting new values.

For example, if the value is always in the memory of the server process, then it could be requested from there whenever it is asked for. If the value is read from a remote device but does not change frequently, then it can be read once and cached. If the value is read from a remote device but changes, then it might be the only way to read it every time from the remote location. However, if there is a large amount of tags to be read and multiple clients, then the application developer might want to have a short-term cache where the values are read for a certain period of time to relieve the stress on the network.

The data mapping mechanism is not dependent on the type of an OPC UA object but on the individual object itself. Therefore it is required that data mapping implementation should be separated from the types.

3.3.3 Type information on the client-side

OPC UA client developers would benefit if the type information from the server address space would be available when the client application is developed. For example, server data could be organized into native Java objects and those could be used in UI applications as models in the common Model-View-Controller (MVC) architecture.

Currently, the Prosys OPC UA Java SDK supports caching the server address space on client-side (Figure 13). However, this is limited only to caching the nodes as the basic node classes. It would be useful if the data from the server was organized as whole OPC UA objects instead of individual nodes. This way, the client developers could build their programs on a higher abstraction level. Each object could be a data source for a UI component, for example.

To access the child nodes of an object, the clients need to use the Translate-

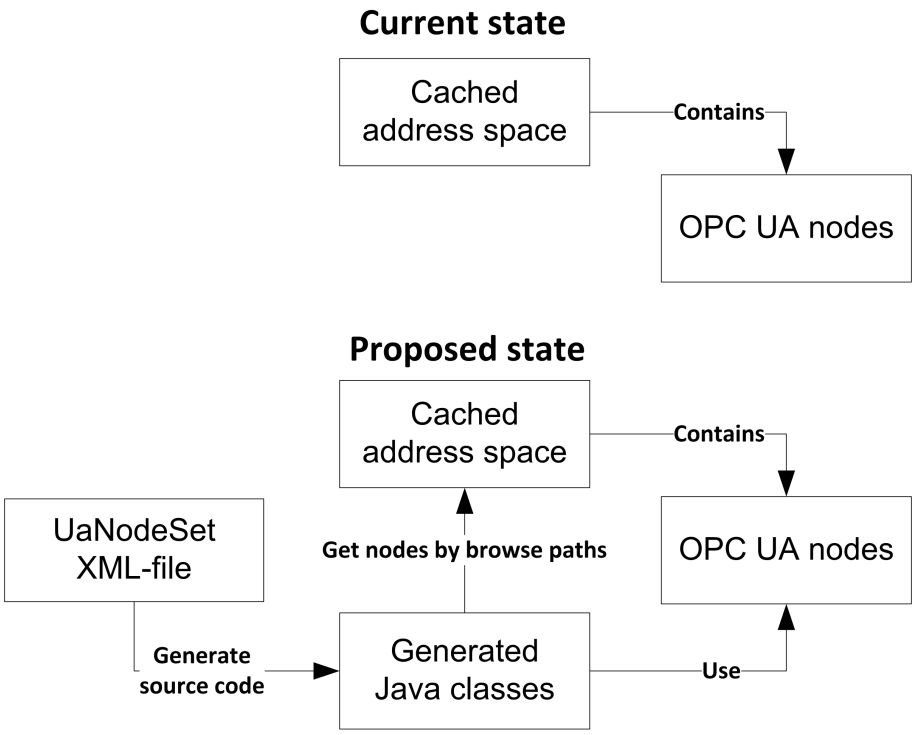

Figure 13: Overview of the requirements on the client-side.

BrowsePathsToNodeIds-service. This service returns the node identifiers for the child nodes, but the client has to provide all the browse paths of the child nodes. Instead of having to manually provide those browse paths, either source code could be generated for them or the type address space could be read to construct them automatically.

One restriction to using UA data as objects on client-side is that the service calls are designed to be used as mass operations. Therefore the client should not request data for single objects separately, but instead for all the objects that are being used on a specific time. It is also possible that a client is connected to multiple UA servers that represent the same address space. Therefore it should be made possible to abstract away the servers from the address space.

3.3.4 Custom data types

Each OPC UA variable has a data type for its value. The OPC UA specification provides standard data types, such as Boolean, DateTime, Double and String, which are sufficient for most use cases. However, some use cases require using custom Enumerations and Structures.

If UA application developers want to use custom Enumerations or Structures

with the SDK, they need to first define the data types in the address space of the server, then create Java classes for the data types and finally create custom encoders and decoders for structured data types. The encoders and decoders are used when the structures are transmitted over network. The last two steps are generated for the UA Java Stack, but the code generator does not support the UaNodeSet-format as input. Thus, the new code generator should also generate the necessary code for data types.

When custom data types are used, the encoders and decoders have to be available on both client- and server-side. At the moment, this is achieved by sharing source code when the applications are developed, which is not feasible when the client and the server are developed independently of each other.

The OPC UA specification defines *OPC binary type dictionaries* (OPC Foundation 2012b) that make encoding information available at the OPC UA server address space. This way, clients do not need to know how to encode and decode different data types beforehand. However, the type dictionaries are not currently supported by any major SDKs, so OPC UA clients cannot be assumed to be able to use them at the moment. Thus, initial support for source code sharing option would at least make it possible to use custom structure data types when the server and the client are developed together.

3.4 Scope of the thesis

It would not be feasible to discuss all the requirements more deeply in this thesis. Therefore only the type instantiation algorithm and a general source code generator are designed in the next sections. The type instantiation algorithm is important for code generation, because the generated object type classes will require that the nodes of the object type exist in the server address space. The general source code generator can be used as a base for all the source code generation requirements in this section.

4 Type instantiation

With type instantiation, the users of the SDK should be able to create instances of the types specified in the server address space. These instances are later on used in conjunction with generated code. In this section, first an overview of the designed type instantiation algorithm is given. Then, all the steps of the algorithm are described individually. Finally, an example of how the instance is used from a Java class is given.

The design of the type instantiation algorithm was done by implementing a functional prototype and considering how it worked. The prototype could take a type node from the server address space and create an instance of that type and all its mandatory nodes. This ensured that most of the details needed for the design were taken into account.

4.1 Overview

Type instantiation algorithm is described in the OPC UA Specification Part 3: Address Space Model (OPC Foundation 2012b). However, specification leaves a lot of open-ended questions of the implementation of the algorithm (Table 5). To be able to design the type instantiation algorithm, those questions need to be answered.

First, it is decided that an instance declaration node should not be instantiated as multiple nodes even if it had multiple browse paths. If the designer of the type wants to have separate nodes, she can always create separate instance declaration nodes. Second, it is decided that new nodes are always created when a type is instantiated. Later on, the algorithm can be extended to use already existing nodes for specified instance declarations. Third, non-hierarchical references are also copied to the instance. In future, this might depend on the modeling rules given for the instance declarations.

Proposed type instantiation algorithm consists of five steps (Figure 14). First, instance declaration hierarchy of each individual type is constructed. Second, instance declaration hierarchies of the subtype and the supertype have to be merged to a fully inherited instance declaration hierarchy. Third, browse paths are replaced with references to the corresponding instance declarations. Fourth, all mandatory instance declarations are instantiated. Fifth, optional instance declarations can be optionally instantiated.

The designed algorithm takes for granted that the server type address space can be read in its abstract form. Loading of the type address space to the memory of

Table 5: Comparison of the OPC UA Specification (OPC Foundation 2012b) and proposed type instantiation algorithm.

Specification	Proposed design
"Multiple BrowsePaths to the same Node shall be treated as separate Nodes. An Instance may provide different Nodes for each BrowsePath."	If a node has multiple browse paths, then only one node is created during the instantiation. With this approach, multiple browse paths to the same node are treated as a single node.
"The Nodes within the newly created hierarchy may be copies of the InstanceDeclarations, the InstanceDeclaration itself or another Node in the AddressSpace that has the same TypeDefinitionNode and BrowseName."	If the instance declaration is the type declaration, an object or a variable, then a new copy of the instance declaration shall be created. If the instance declaration is a method, then the instance declaration itself is used.
"Note that the ModellingRules defined in this standard do not define how to deal with non-hierarchical References between InstanceDeclarations, i.e. it is Server-specific if those References exist in an instance hierarchy or not."	Non-hierarchical references to other instance declarations are copied to the instance hierarchy. Non-hierarchical references to other nodes than instance declarations shall have same target nodes as defined in the instance declaration node.

the OPC UA server has been implemented in the SDK previously (Palonen 2010). Thus, the type address space is represented just with the standard notation in this section.

4.2 Creating an instance declaration hierarchy

An example of two type definitions are taken to demonstrate the type instantiation algorithm (Figure 15). First, the instance declaration hierarchy of the AlphaType is created. This means collecting all the relevant instance declarations and references in the type definition hierarchy.

The initial instance declaration hierarchy consists of a set of temporary instance declarations. The declarations are called temporary because they are replaced with permanent declarations in the third step of the algorithm. Each temporary instance declaration has a set of browse paths (Table 6), a set of *internal* references and a set of *external* references (Table 7). Internal references target other temporary

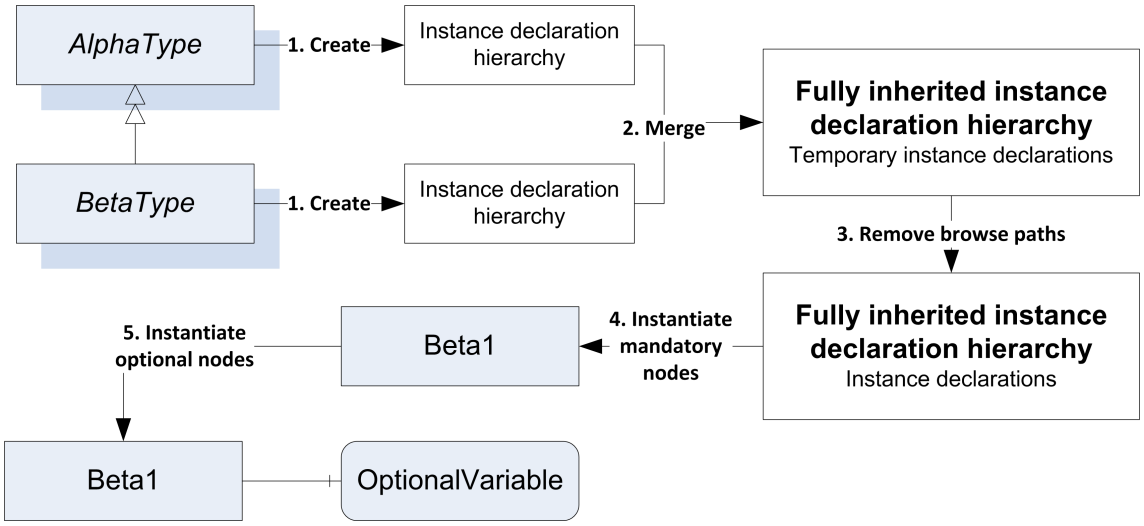

Figure 14: The designed type instantiation process.

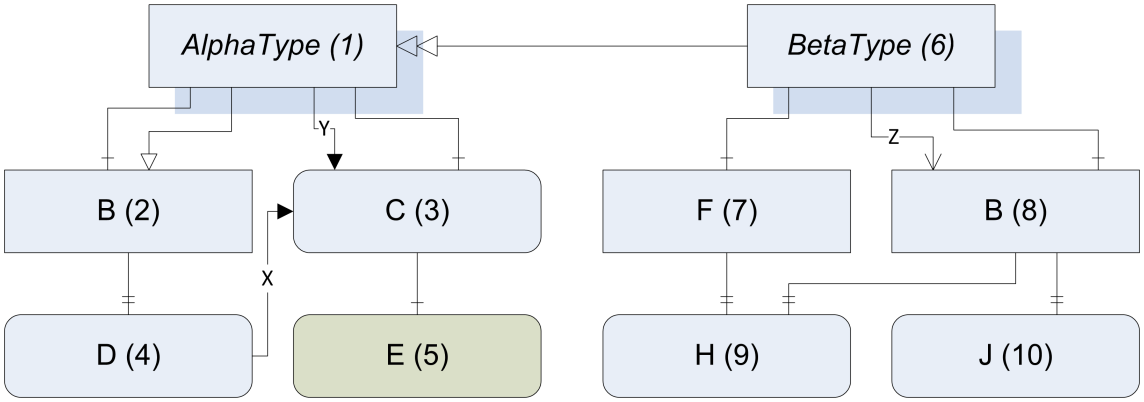

Figure 15: Example type hierarchy. Similar to the one in the OPC UA specification (OPC Foundation 2012b), but certain contradictions were fixed. Node identifiers are shown inside parentheses and the E-node which has no modeling rule is marked with different color. Reference types X and Y are non-hierarchical, whereas Z is hierarchical.

instance declarations. Their targets are represented as browse paths, because it is not yet known which instance declaration will really be the real target of the reference. The target can be overridden in a subtype. External references, e.g. the HasTypeDefinition-references, are references to nodes outside the type definition. They are always non-hierarchical.

It should be noted that neither the E-node nor the reference from C to E is added to the instance declaration hierarchy. E-node is just part of the type hierarchy and should not be added to the instances.

The form of the instance declaration hierarchy defined here differs from the one

Table 6: Temporary instance declarations of the AlphaType.

Node identifier	Browse name	Browse paths
1	AlphaType	/
2	B	/B
3	C	/C
4	D	/B/D

Table 7: References of the temporary declarations in the AlphaType. HasModellingRule-references are omitted here, because it is customary to not include them in the instances of types.

Declaration	Reference type	Target
AlphaType (1)	HasComponent	/B
AlphaType	HasNotifier	/B
AlphaType	HasComponent	/C
AlphaType	Y	/C
B (2)	HasProperty	/B/D
B	HasTypeDefinition	BaseObjectType
C (3)	HasTypeDefinition	BaseVariableType
D (4)	X	/C
D	HasTypeDefinition	PropertyType

defined in the OPC UA specification (OPC Foundation 2012b). In the specification, each browse path is represented as a single instance declaration. The same approach would not be a good one in the design represented earlier, because it defined that if a node has multiple browse paths, it should still be considered a single instance declaration (Table 5).

4.3 Merging instance declaration hierarchies

The instance declaration hierarchy formed for the AlphaType would suffice for creating instances. To create instances of BetaType, similar instance declaration hierarchy has to be created and merged with the instance declaration hierarchy of the AlphaType. In the BetaType, the H-node has two browse paths (Table 8). Both paths appear in the internal references (Table 9).

Each temporary instance declaration of the AlphaType is compared to the dec-

Table 8: Temporary instance declarations of the BetaType.

Node identifier	Browse name	Browse paths
6	BetaType	/
7	F	/F
8	B	/B
9	H	/F/H, /B/H
10	J	/B/J

Table 9: References of the temporary declarations in the BetaType.

Declaration	Reference type	Target
BetaType (6)	HasComponent	/B
BetaType	Z	/B
BetaType	HasComponent	/F
F (7)	HasProperty	/F/H
F	HasTypeDefinition	BaseObjectType
B (8)	HasProperty	/B/H
B	HasProperty	/B/J
B	HasTypeDefinition	BaseObjectType
H (9)	HasTypeDefinition	PropertyType
J (10)	HasTypeDefinition	PropertyType

larations of the BetaType. If the BetaType does not have a temporary instance declaration with any of the browse paths the declaration in question has, then the declaration can be added to the BetaType. Otherwise the declaration is merged with the existing declaration.

Declarations are merged by combining the browse paths and the references of the declarations. Combining the browse paths is simple, but the references requires more processing, because their equivalence is not so straightforward.

Usually a reference is overridden, if a reference in the subtype has the same source and the target and its type is the same or a subtype. However, this does not apply to all non-hierarchical references (Table 10). For example, a HasTypeDefinition-reference can be overridden even when it has a different target in the subtype, because each node is allowed to have only one HasTypeDefinition-reference. It is also possible to have multiple non-hierarchical references between two nodes with identical types. Whether to allow multiple references or not can be determined for

each standard non-hierarchical reference type, but for custom reference types only a default handling can be done.

Table 10: Summary of the rules for merging the references. Types are considered same if they are same or if either is a supertype of another.

Hierarchical	Same type	Same target	Action
✓	✓	✓	Override
✓	✓	–	Merge
✓	–	✓	Merge
✓	–	–	Merge
–	✓	✓	May override
–	✓	–	May override
–	–	✓	Merge
–	–	–	Merge

The only overridden instance declaration in the BetaType is the B-node (Table 11). C- and D-nodes are copied from the AlphaType. The type definition node always overrides its supertype and gets all its references (Table 12). B-node gets references from the node it overrides, except the HasTypeDefinition-reference, because it is unique and already present in the BetaType. Other references are copied from the AlphaType.

Table 11: Temporary instance declarations of the fully inherited BetaType. Nodes from the AlphaType are highlighted.

Node identifier	Browse name	Browse paths
6	BetaType	/
7	F	/F
8	B	/B
9	H	/F/H, /B/H
10	J	/B/J
3	C	/C
4	D	/B/D

After merging, a fully inherited instance declaration hierarchy of the BetaType has been formed (Figure 16). If the type hierarchy had more types than two, then the instance declaration hierarchy of the next subtype could be merged with the

hierarchy of the BetaType. Thus, creation of fully inherited instance declaration hierarchies is possible with this algorithm, in general.

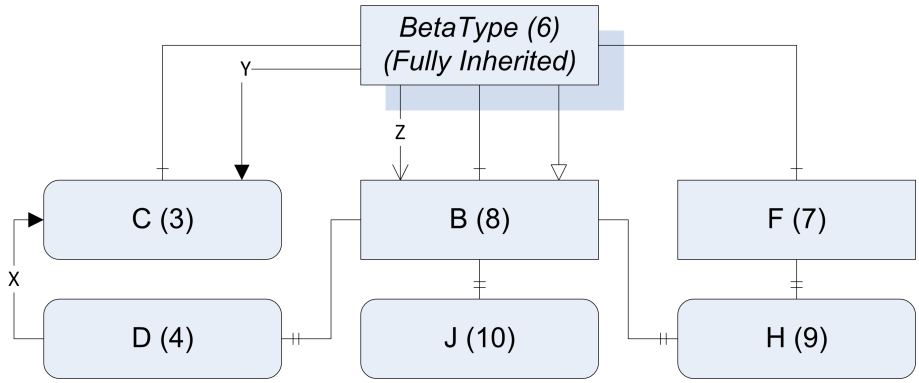

Figure 16: Fully inherited instance declaration hierarchy of the BetaType.

Table 12: References of the temporary declarations in the fully inherited BetaType. References from the AlphaType are highlighted.

Declaration	Reference type	Target
BetaType (6)	HasComponent	/B
BetaType	Z	/B
BetaType	HasComponent	/F
BetaType	HasNotifier	/B
BetaType	HasComponent	/C
BetaType	Y	/C
F (7)	HasProperty	/F/H
F	HasTypeDefinition	BaseObjectType
B (8)	HasProperty	/B/H
B	HasProperty	/B/J
B	HasProperty	/B/D
B	HasTypeDefinition	BaseObjectType
H (9)	HasTypeDefinition	PropertyType
J (10)	HasTypeDefinition	PropertyType
C (3)	HasTypeDefinition	BaseVariableType
D (4)	X	/C
D	HasTypeDefinition	PropertyType

4.4 Replacing browse paths

After all the instance declaration hierarchies have been merged, the browse paths are not needed anymore. It makes further type instantiation process easier if the references to browse paths are replaced with references to instance declarations. In this part of the algorithm, the temporary instance declarations are converted to instance declarations by converting the internal references to *instance* references.

Converting the references is straightforward. The target browse path of each internal reference is replaced with the corresponding instance declaration. Comparison of the temporary and the final instance declaration is presented in Table 13. The actual node in the type address space is still needed for the attributes and the modeling rule of the instance declaration.

Table 13: Comparison of the temporary and the final instance declaration.

Field	Temporary	Final
Node	✓	✓
Browse paths	✓	–
Internal references	✓	–
Instance references	–	✓
External references	✓	✓

4.5 Instantiating mandatory instances

In this step of the instantiation, an actual instance of a type is created. Node identifiers need to be supplied for all the nodes of the new instance. Since the type of the node identifiers and the method for generating them is server specific, it is taken for granted in this algorithm that node identifiers are available. Also display names and descriptions for the new nodes can be supplied for the algorithm, but they can be given to the nodes after the instantiation too.

First, the corresponding instance node of the type definition node is created. The instance node shall copy all the relevant attributes from the type node. Second, hierarchical instance references are followed from the type definition node instance declaration. If the next instance declarations are mandatory, they are created too. Finally, instance references are again followed from the mandatory instance declarations. Since the instance reference hierarchy can contain loops, it should be checked that an instance node has not been created before creating it again. If an instance

node is already created, the algorithm shall stop processing the following instance references.

After all the hierarchical instance references have been processed, then all the mandatory nodes have been created. Next, instance references are instantiated. Each instance reference of a created instance node is instantiated, if the target of the reference has been instantiated. This way, only references to other mandatory nodes are created. Finally, external references are instantiated for the created instance nodes.

4.6 Instantiating optional instances

Instantiation of optional instances is similar to instantiation of the mandatory instances. Difference is that the instantiation does not start from the type definition node but from the optional instance declaration node that is instantiated. Then again hierarchical instance references are followed and mandatory instance nodes are created if they have not been created before. Thus, the algorithm needs to know corresponding instance nodes and instance declarations.

After all the mandatory nodes connected to the target optional node are created, then all created nodes are iterated over again to create possibly missing instance references. Some instance references might not have been created for the mandatory nodes, because the optional node was not present at the time of instantiation of mandatory nodes.

In addition to the Mandatory and the Optional modeling rules, OPC UA specification defines OptionalPlaceholder, MandatoryPlaceholder and ExposesItsArray modeling rules. Placeholder modeling rules mark instance declarations that can be present in instances, but with custom browse names. Thus, the instantiation algorithm for optional instances can be used for placeholders too, providing that also new browse names are supplied for the algorithm. In addition, it should be forced that mandatory placeholders are instantiated before the instance is made available to clients. Instances that expose their arrays should programmatically call the instantiation algorithm when their array value changes. Custom behavior might have to be coded to define how to map the values of the array to the nodes of the address space.

4.7 Using the instance

So far the algorithm has made the structure of the instance right in the address space. The next step is to make this structure information available to the IDE which programmers use to write OPC UA server software. The structure information can be read by IDEs from Java classes that have the same structure as the OPC UA types.

The Java class that represents the BetaType accesses its components by following its component references 1. The type assumes that the components exist already. This differs from the previous handwritten classes and other code generators, where the instantiation was done separately in each class and the components were accessed through member variables. Separate instantiation works for single classes, but when components are overridden, it becomes complicated.

```
public class BetaType extends AlphaType {
  public BaseObjectType getB() {
    return getComponent("B");
  }

  public BaseObjectType getF() {
    return getComponent("F");
  }

  public BaseVariableType getC() {
    return getComponent("C");
  }
```

Listing 1: Sketch of a class to demonstrate how the instances of the BetaType could be used.

It is questionable whether the BetaType class should contain getters for nodes such as /B/D and /B/H. These are not properties of the BetaType but of its B-component. However, now those properties cannot be accessed with exact Java methods, because BaseObjectType does not contain information about them. If the B-component had an object type which defined the D- and H-properties, then they could be accessed, e.g. with code `getB().getD()`. Thus, the maker of the information model can decide whether to leave the properties inaccessible directly or not.

These Java classes could be written by hand, but to reduce manual workload, the classes should be generated based on the type information. This is done with source code generation in the next section.

5 Source code generation

In this section, a design for a generic source code generator is presented. The design is applicable for most code generation requirements defined in Section 3, such as server- and client-side type information and custom data types. However, discussing the details of any individual requirement was not considered necessary in the scope of this thesis. In this section, the code generator is presented to produce Java classes for OPC UA object types.

The design was formed by building a functional prototype in conjunction with the design. This ensured again that any details needed for the design were taken into account. First off, UaNodeSet XML-files were preprocessed so that the data would be easier to use for the code generation. Then, the handwritten classes in the Prosys OPC UA Java SDK were used as the basis for prototyping the actual code generator. The final prototype could produce usable Java classes of the object and variable types.

5.1 Overview

Three code generation methods described by Völter (2003) were introduced in the Background section: templates + filtering, templates + metamodel and API-based generators. In this section, it is decided which method is the most suitable for the use case in this thesis. The handwritten source code files that existed already in the Prosys OPC UA Java SDK were examined to understand what kind of source files the generator should generate and thus what properties the use case has.

The main benefit of the API-based generator would be that the produced source code would always have a valid syntax. In contrast, writers of the templates can do unintentional syntax errors which are not evident until the code is first generated and after that compiled. However, API-based generator method is not feasible when the size of the source code files is large, which was the case in the handwritten source code files. It is hard to see the resulting source code from the program that uses the API-based generator. The template methods were also considered easier to use and modify.

When the first prototypes of the generator were made, it became soon clear that the UaNodeSet XML-files need to be processed quite much to get enough information for the code generation. Thus, simple templates + filtering method was not possible because of the structure of the model and it was decided to use the templates + metamodel method.

For templates, Mustache (Wanstrath 2013) templates were chosen. Self-made and Turing-complete template languages were also in consideration. Self-made template languages would be appropriate for simple search and replace code generation. However, the use case required more sophisticated template features which exist in the readily available template languages. Fleet (Ablamonov 2013) template language was used for the first prototypes. Fleet allows using all the statements of the programming language inside the templates. This made the templates harder to read, so finally Mustache was used, because it forces clear templates with simple syntax, but does not restrict templates too much for the use case.

The overview of the code generation infrastructure is depicted in Figure 17. Clojure (Hickey 2013) programming language was used to write the generator program prototype, because it allowed flexibility for processing the model XML-files and is also interoperable with Java. In addition to the UaNodeSet files and the templates, the generation process also requires configuration data.

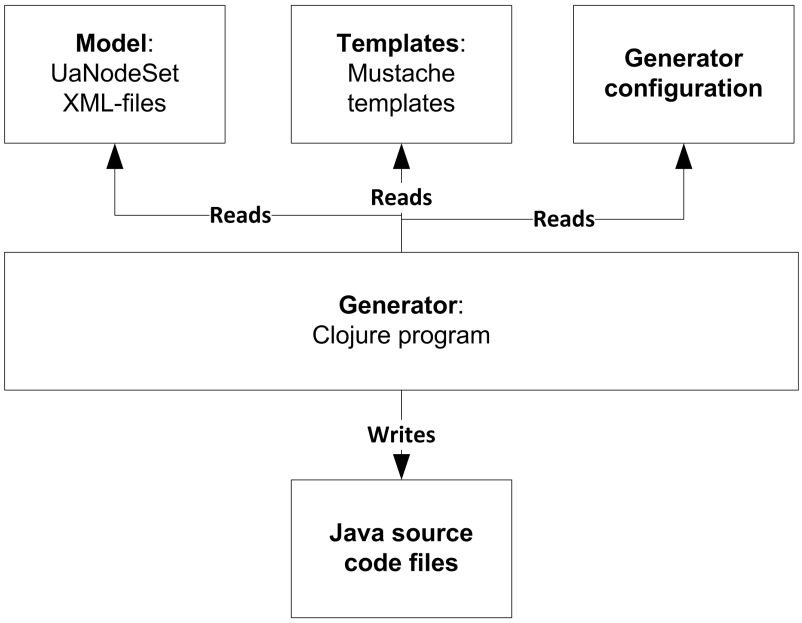

Figure 17: Overview of the source code generator and its data sources.

5.2 Model structure

UaNodeSet XML-files represent the actual nodes in the address space. They are used to store information models by containing the type nodes of the address space and in addition the standard instance nodes that are defined in the OPC UA specification (OPC Foundation 2012d). In general, they can be used to serialize or persist the

address space or some parts of it.

To make the size of the UaNodeSet files smaller, the namespaces of the node identifiers and the browse names are represented in numeric form. The namespace number corresponds to a namespace URI introduced in the namespace URI array in beginning of the file. For example, the node identifier "ns=1;i=4001" in Listing 2 has namespace index 1 and an identifier part 4001. The namespace index corresponds to the PLCOpen namespace. Node identifiers in the file can be aliased to make them human readable. All the aliases are also defined in the beginning of the file. An example of that in the Listing 2 is the HasSubType-reference.

```
<?xml version="1.0" encoding="utf-8"?>
<UANodeSet>
  <NamespaceUris>
    <Uri>http://PLCopen.org/OpcUa/IEC61131-3/</Uri>
    <Uri>http://opcfoundation.org/UA/DI/</Uri>
  </NamespaceUris>
  <Aliases>
    <Alias Alias="Boolean">i=1</Alias>
    <Alias Alias="SByte">i=2</Alias>
    <Alias Alias="Byte">i=3</Alias>
    ...
    <Alias Alias="HasSubtype">i=45</Alias>
    ...
  </Aliases>
  <UAReferenceType NodeId="ns=1;i=4001"
                   BrowseName="1:HasInputVars">
    <DisplayName>HasInputVars</DisplayName>
    <References>
      <Reference ReferenceType="HasSubtype"
                 IsForward="false">i=47</Reference>
    </References>
    <InverseName>InputVarsOf</InverseName>
  </UAReferenceType>
  ...
</UaNodeSet>
```

Listing 2: Example of a UaNodeSet file, part of PLCOpen information model (OPC Foundation and PLCopen 2010).

The rest of a UaNodeSet file is a list of the nodes in the address space. Each node definition is an XML element that has a tag name which corresponds to the node class of the node. The attributes of the node are represented as attributes or child elements of the XML element. The references of the node are represented

as a list of Reference-elements which contain the reference type and target as node identifiers. The Reference-elements can also indicate that the reference is an inverse reference by setting the IsForward-attribute false, meaning that the node itself is the target of the reference.

5.3 Mustache templates

Mustache template language uses a hash table as an input model. The hash table contains keys which are names of the values. The names are used in the templates, and when the templates are applied, the names are replaced with the values in the hash table. A value can be either a text, a hash table, a list, a boolean or a lambda function. Texts are used for simply replacing the names in the templates. Inner hash tables allow structuring of the data as trees. Lists can be iterated in the templates and an inner template can be applied for each element in the list. Booleans can be used to conditionally include some text. Lambda functions are useful for simple operations such as capitalizing text.

An example of a mustache template is given in Listing 3. All the mustache tags are separated from static text with double curly braces. First, a simple text replacement is done with the tag `{{package}}` for defining in what package the file resides. Next, with the `{{#dependencies}}` tag, a list of dependencies is iterated and for each dependency, an import line is rendered. The elements of the list are hash tables which contain keys "package" and "class". Thus, the `{{package}}` tag inside the iteration does not necessarily have the same value as the similar tag outside the iteration. The ending of the inner template is marked with a closing tag `{{/dependencies}}`.

```
package {{package}};

import {{package}}.{{classname}};
{{#dependencies}}
import {{package}}.{{class}};
{{/dependencies}}
```

Listing 3: Beginning of a Java class in a Mustache template.

Hash tables, booleans and lambdas are used with same notation as lists. For hash tables, the inner template is applied with the inner hash table as the model. For booleans, the inner template is applied only if the boolean value is true. Inverse handling of booleans is also possible. For lambdas, inner template is given as an input value before applying the template and the result will be the return value

of the lambda function. Lambda function can itself apply the inner template if necessary.

5.4 Template structure

The generated source code needs to be integrated with the non-generated source code in such a way that regeneration does not lose any non-generated code. In this thesis, it was also required to generate the source code to replace already handwritten source code files. This causes an additional requirement, because the handwritten source code has been part of a published public API which should not change because this would affect the users of the SDK. One part of the API needs to be generated, but another part should remain handwritten.

It is not possible in Java programming language to implement a class in two separate files. One way to overcome this is to mark the generated code with comment blocks in the source code file. The generator would generate these blocks, but leave the rest of the file untouched, thus preserving any handwritten code outside the generated blocks. Other way, which is used by the C++ generator in the UaModeler, is that the generated code resides in a base class and handwritten code is placed in a subclass. This makes the type hierarchy more complex, but the integration of generated code simpler. It also makes it possible to override generated code with handwritten code.

Since it was not known whether the overriding of generated code would be necessary or not, the integration was done with generated base classes and handwritten subclasses. However, the requirement concerning the API applies only to the existing handwritten Java classes, and later on, another integration method could be used for generated source code that has no public API yet.

To separate the generated and handwritten classes, it was designed that the generated class names shall be appended with ”Base” (Table 14). In addition, the generated classes are put into a different Java package by appending the original package with ”.base”. This way, the generated code stays out of sight, which is often desired because the generated code is not usually read or written by the developer.

In the scope of this thesis, only the template for object types was studied more rigorously. Variable types are similar to the object types which have no methods or child objects and data types should resemble the classes used in the Java stack. The generated base class for object types shall include:

Constructors that simply relay the constructor call to the super class.

Table 14: Comparison of the generated base class and the handwritten subclass for object types.

Base class	Subclass
Generated many times	Generated once
Name is appended with ”Base”	Has name of the type
Package is appended with ”.base”	Has package based on the namespace
Extends from supertype	Extends from own class appended with ”Base”
Getters and setters for components and properties	–
Stubs for method calls	Implementation of methods
No custom behavior	Custom behavior can be added

Getters and setters for variables and objects. In addition, getters and setters for the values of variables can be available too.

Stubs for method calls which handle the checking of node identifiers of the methods and relay method calls to corresponding abstract Java methods. The methods are implemented in the subclass.

When generated for the first time, the subclass shall include method stubs that correspond to the abstract ones in the base class.

5.5 Generator architecture

The generator program consists of two parts: the parser and the generator (Figure 18). The parser is responsible for preprocessing the UaNodeSet files and creating a node identifier index. The generator creates instances of metamodels based on the data the parser provides, and then applies the Mustache templates to the instances of the metamodels.

The parser preprocesses the UaNodeSet files so that the namespace indexes are replaced with the real namespace URIs in the node identifiers and the browse names. This has to be done because the namespace indexes are specific to the UaNodeSet files, but all the files are used together when node identifiers are searched from the node identifier index. The parser has to replace the aliases with the node identifiers first, because the aliases are also specific to the UaNodeSet files.

The parser creates a node identifier index. The index can be used to find out the corresponding XML element of a node identifier. With the node identifier index, the

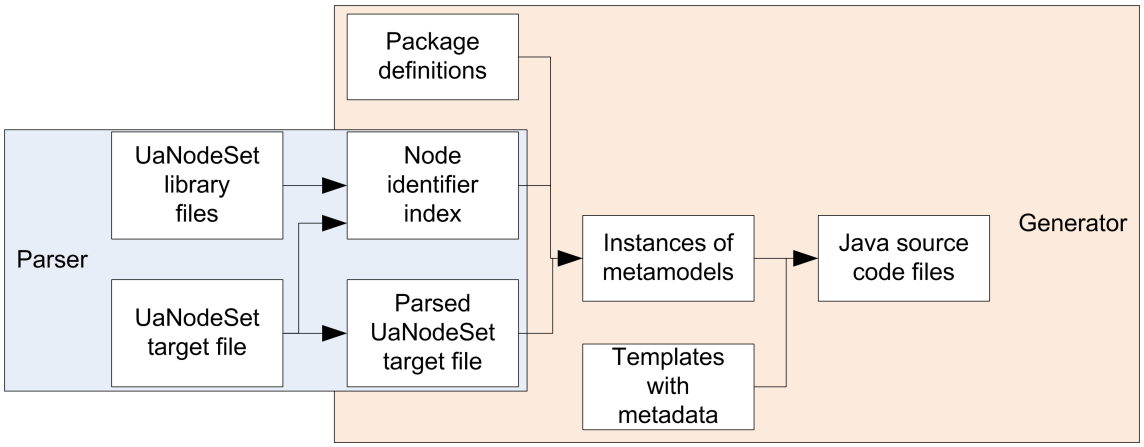

Figure 18: The generator program architecture. The program consists of two parts: the parser and the generator. Boxes and arrows illustrate the data flow through the generator.

generator can take the target node identifier of a reference and then transform the identifier to a corresponding XML element. This way, the components, properties and methods of an object type can be found during construction of the metamodels.

The target UaNodeSet file is the model that is used in source code generation. Other UaNodeSet files are needed just to build the node identifier index. The generated source code expects that all other types that it uses have been previously generated and can be found in Java packages defined in the package definitions.

The generator takes four data sources: the parsed UaNodeSet target file, node identifier index and package definitions. From these, the generator creates the instances of the metamodels. There is one metamodel for each node class. Then, the generator applies the templates to the instances of the metamodels, producing the Java source code files.

The package definitions map the namespaces of the UA nodes to Java packages. This way, the generator knows where the dependencies of a generated Java class can be found and then the import statements can be added to the beginning of the Java class. The OPC UA Java Stack (OPC Foundation 2013a) organizes its types under several packages based on the supertypes of the types, so the final mapping is done from a namespace and a supertype to a Java package. For example, standard data types with Structure or Enumeration as their supertype reside in the `org.opcfoundation.ua.core` package, but other data types reside in the `org.opcfoundation.ua.builtintypes` package.

The generation process is the following (Figure 19). First, the generator looks at the template directory and loads all the templates. Then, each template (1.) is

searched from the template data where it is defined that which XML elements the template is applied to. Next, those XML elements are read from the parsed target UaNodeSet file and for each element (2.), an instance of a metamodel is formed (3. and 4.). The instance is applied to the template which is currently being processed (5.). The final result is a Java source code file.

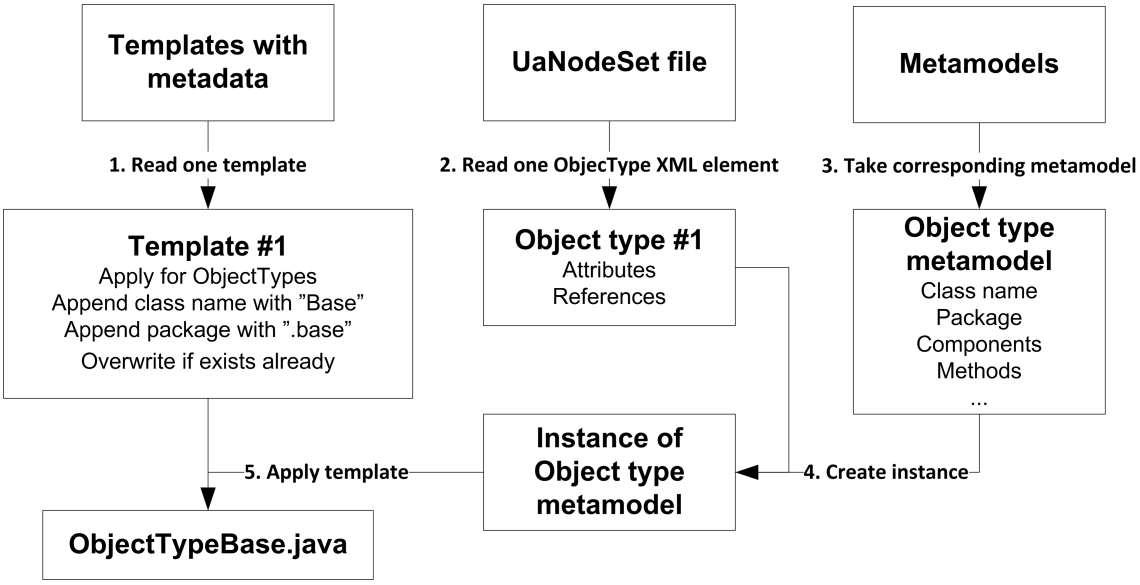

Figure 19: Demonstration of the relations between templates, UaNodeSet files and metamodels. The steps of the generation process are also shown.

5.6 Structure of the metamodels

The values in the metamodels should be almost so fine-grained that the templates need to just reference the correct values by name in the model. However, the metamodels should be also so general that creating new kinds of source code files should not require editing the metamodels, but instead the templates. The metamodels for the generator were formed by incrementally building the templates for the already existing handwritten type classes in the SDK. If some information was needed, it was added to the metamodels.

The metamodels are used as Mustache data sources, so they should be hash tables. To provide all the information needed for the object type template described in Section 5.4, the following data sources need to be used:

- information taken from the XML element directly
- information taken from XML elements accessible through the references or other node identifiers in the XML element

– some parts require a mapping from OPC UA to Java, such as namespaces to packages and some OPC UA types to Java primitive types

When node identifiers are followed, the resulting XML element can also be transformed into an instance of a metamodel. However, the node identifiers are usually followed only once or twice and, e.g., the components of a component need not be known in the metamodel (Figure 20).

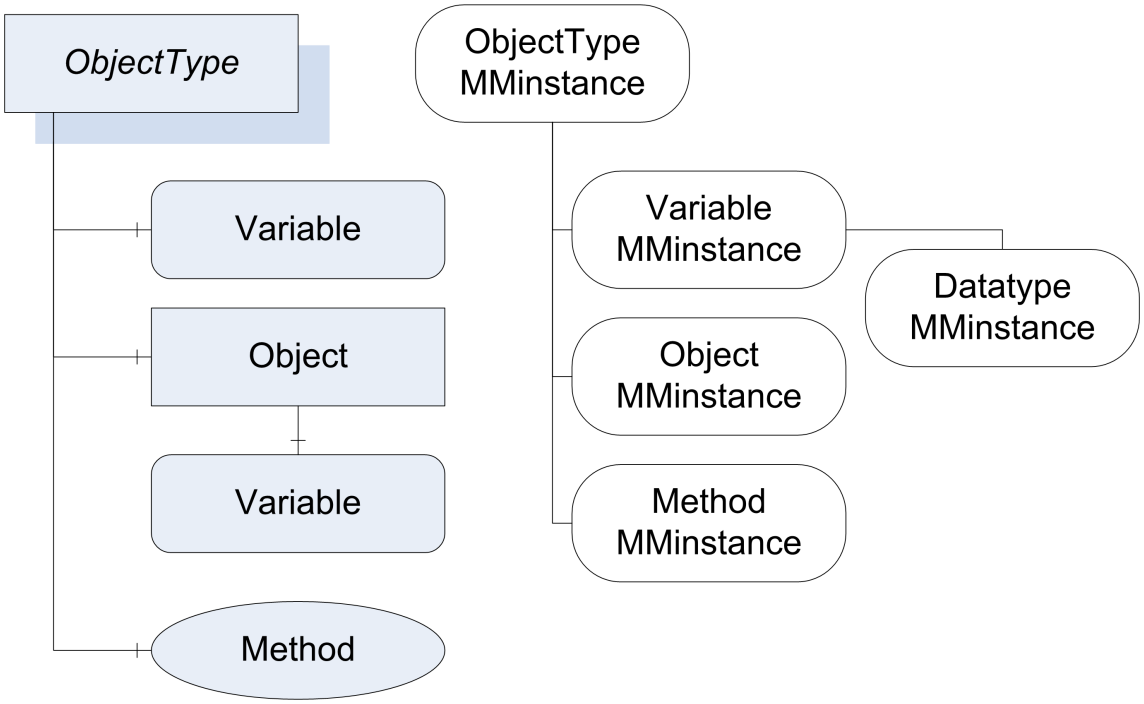

Figure 20: A metamodel instance created from an example object type. Metamodels of components are also created, but not of components of components. Still, a metamodel instance of the variable datatype is created.

The metamodels do not contain exactly the same information that is defined for each node class. Some of the information needs to be processed further to fulfill the needs of the templates. For example, OPC UA variables have a value rank, which is a numerical representation of the fact whether the value is an array and how many dimensions it has. This numerical representation is transformed so that it can be presented as is required in Java.

5.7 Applying the templates

After an instance of a metamodel is created, the applying process is straightforward. The instance is fed to Stencil (Santiago 2013), a Clojure implementation of Mus-

tache. Stencil applies a template to the instance, resulting in Java source code. An example of this process is given in Appendix A.

In Java, each dependency has to be introduced by an import statement at the beginning of the file. The template data contains information about what kind of dependencies are required for each template. For example, an object type template can define that it requires dependencies to its variables. However, there are situations when the template does not use all the dependencies. Usually it is desired that any unused import statements are removed. This is achieved by removing them after a template is applied. An unused import can be detected by searching the file for direct references to the classes in the import statements. If any direct references cannot be found, the import is unused.

When the Java source code files are saved, some templates require additional information that is not present in the UaNodeSet files. An example of this is the generation of the base class, when it is required that the class name is appended with ”Base”. This information is not naturally found from UaNodeSet files and is actually dependent on the template file. Therefore it is required that templates contain metadata about possible modifications to the class and package names. In addition, in the template metadata it is defined whether or not any existing Java source code files should be overwritten (Figure 19).

Finally, the generator creates the directory structure corresponding to Java package names and puts the final source code files into those directories. By doing this, there is no need to manually move files to different folders after the code has been generated. This enables tighter integration with the build process of an OPC UA application.

6 Conclusions and future work

Information modeling capabilities in the OPC Unified Architecture are essential for interoperable software development. Actually using the capabilities is not straightforward until a substantial development effort has been made for building a higher level framework with which the developers can make use of the OPC UA information models in their programs. In this thesis, requirements and design principles for such a framework have been created.

6.1 Answering the research questions

First research question in this thesis was: *"What are the requirements for source code generation from OPC UA information models?"* Requirements included having the type information available when developing OPC UA applications on both client- and server-side and being able to use custom structured data types. When compared to other source code generation tools for OPC UA, the results of this thesis have a clear distinction: the separation of type instantiation and using the types. The distinction makes development of the type instantiation algorithm easier, since the algorithm is not applied during code generation, but instead during runtime. Another distinction to previous implementations was that the data mapping implementation should be separated from the generated type information, but the requirement was not studied any further in this thesis.

Second research question was: *"How should the generated source code be used in OPC UA applications?"* As suggested in the requirements, the type instantiation algorithm was implemented separately of the source code generation process. This lead into a detailed design of the type instantiation algorithm itself, because the generated source code still needed to use the instantiated structures.

Third research question was: *"How should the source code generation be done in practice?"* A design for the source code generation tool was proposed in this thesis. The design took note of several practical issues including namespace handling and other source code generation requirements.

6.2 Future work

The type instantiation algorithm and the code generation system described in this thesis were implemented as prototypes. These prototypes are used as a basis for further development and the results will be released as part of the Prosys OPC UA

Java SDK. The final implementations can be validated to fulfill the requirements presented in this thesis. In addition to technical details, it is important to design how the developers are expected to use the system. This will be taken into account when the work is integrated to the product.

Out of all the requirements listed in Section 3, just type instantiation and source code generation for server-side were discussed in this thesis. For instance, data mapping and source code generation for client-side and custom data types are also vital requirements but did not fit into the scope of this thesis. Full-fledged type information usage would require implementation of those requirements too.

It shall take time until the OPC Unified Architecture is widely adapted and all the features in the specification are implemented by the tools available in the market. This thesis can be seen as a milestone in that process.

References

Ablamonov, Illia (2013). *Fleet – Templating System for Clojure*. URL: https://github.com/Flamefork/fleet (Last accessed July 24, 2013).

Armstrong, Randy (2013). *UA SDK ModelCompiler distribution – topic on OPC Foundation Message Board*. URL: http://www.opcfoundation.org/forum/viewtopic.php?t=4733 (Last accessed Jan. 22, 2013).

Arnoldus, BJ (2011). "Less is more: unparser-completeness of metalanguages for template engines". In: *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE)*, pp. 137–146. URL: http://dl.acm.org/citation.cfm?id=2047887.

CommServer (2013). *OPC UA Address Space Model Designer*. URL: http://www.commsvr.com (Last accessed Jan. 10, 2013).

Goldschmidt, T. and W. Mahnke (2012). "Evaluating domain-specific languages for the development of OPC UA based applications". In: *7th Vienna International Conference on Mathematical Modelling (MATHMOD) Special Session Modelling and Model Transformation in Automation Technologies*. URL: http://seth.asc.tuwien.ac.at/proc12/full_paper/Contribution433.pdf.

Granzer, W. and W. Kastner (2012). "Information Modeling in Heterogeneous Building Automation Systems". In: *Proc. of the 9th IEEE International Workshop on Factory Communication Systems (WFCS' 12)*, pp. 291 –300. DOI: 10.1109/WFCS.2012.6242577. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6242577.

Hickey, Rich (2013). *Clojure programming language*. URL: http://clojure.org/ (Last accessed July 24, 2013).

Hiltunen, Tuomas (2012). "Java Based OPC UA Client Development". MA thesis. Aalto University.

Lehnhoff, Sebastian et al. (2012). "OPC Unified Architecture: A Service-Oriented Architecture for Smart Grids". In: *ICSE 2012 International Workshop on Software Engineering Challenges for the Smart Grid*, pp. 1–7. DOI: 10.1109/SE4SG.2012.6225723. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6225723.

Mahnke, Wolfgang, Stefan-Helmut Leitner, and Matthias Damm (2009). *OPC Unified Architecture*. Springer. ISBN: 978-3-540-68898-3. DOI: 10.1007/978-3-540-68899-0.

OPC Foundation (2011a). *Information Model XML Schema*. URL: http://opcfoundation.org/UA/2011/03/UANodeSet.xsd (Last accessed Jan. 10, 2013).

— (2011b). *OPC UA SDK 1.01*. URL: http://www.opcfoundation.org (Last accessed Jan. 10, 2013).

— (2012a). *OPC Unified Architecture Specification, Part 1: Overview and Concepts, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part1.

— (2012b). *OPC Unified Architecture Specification, Part 3: Address Space Model, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part3.

— (2012c). *OPC Unified Architecture Specification, Part 4: Services, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part4.

— (2012d). *OPC Unified Architecture Specification, Part 5: Information Model, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part5.

— (2012e). *OPC Unified Architecture Specification, Part 6: Mappings, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part6.

— (2013a). *OPC UA 1.02 Java Stack Source Code and Sample Applications*. URL: http://www.opcfoundation.org/DownloadFile.aspx?RI=958 (Last accessed July 5, 2013).

— (2013b). *OPC Unified Architecture for Analyser Devices (ADI) Companion Specification, Release 1.01*. Tech. rep. URL: http://www.opcfoundation.org.

— (2013c). *OPC Unified Architecture for Devices (DI) Companion Specification, Release 1.01*. Tech. rep. Di. URL: http://www.opcfoundation.org.

— (2013d). *OPC Unified Architecture For ISA-95 Common Object Model, Release Candidate 1.01.00*. Tech. rep. URL: {http://www.opcfoundation.org}.

— (2013e). *OPC Unified Architecture Specification, Part 2: Security Model, Release 1.02*. Tech. rep. URL: http://opcfoundation.org/UA/Part2.

OPC Foundation and PLCopen (2010). *OPC UA Information Model for IEC 61131-3*. Tech. rep. URL: http://www.opcfoundation.org.

Palonen, Otso (2010). "Object-oriented implementation of OPC UA information models in Java". MA thesis. Aalto University.

Parr, Terence John (2004). "Enforcing strict model-view separation in template engines". In: *Proceedings of the 13th conference on World Wide Web - WWW '04*, p. 224. DOI: 10.1145/988672.988703. URL: http://portal.acm.org/citation.cfm?doid=988672.988703.

Prosys (2013). *Prosys OPC UA Java SDK*. URL: http://www.prosysopc.com/opc-ua-java-sdk.php (Last accessed Jan. 22, 2013).

Santiago, David (2013). *Stencil – A Clojure implementation of Mustache*. URL: https://github.com/davidsantiago/stencil (Last accessed July 26, 2013).

Sheard, Tim (2001). "Accomplishments and research challenges in meta-programming". In: *Semantics, applications, and implementation of program generation*. Pp. 2–44. URL: http://link.springer.com/chapter/10.1007/3-540-44806-3_2.

HB-Softsolution (2011). *Comet UA Model Designer*. URL: http://www.hb-softsolution.com (Last accessed Jan. 21, 2013).

Unified Automation (2013). *UaModeler*. URL: http://www.unified-automation.com (Last accessed Jan. 10, 2013).

Völter, Markus (2003). "A Catalog of Patterns for Program Generation". In: *Eighth European Conference on Pattern Languages of Programs (EuroPLop)*. URL: http://www.voelter.de/data/pub/ProgramGeneration.pdf.

W3C (Nov. 16, 1999). *XSLT*. URL: http://www.w3.org/TR/xslt (Last accessed July 29, 2013).

Wanstrath, Chris (2013). *Mustache templates*. URL: http://mustache.github.io (Last accessed July 5, 2013).

A Source code generation example

In this section, a source code generation example is presented. First, a template used for generation is shown (Listing A1). Then, an instance of a metamodel is shown (Listing A2). The instance is rendered by applying the template to it. Finally, the resulting generated Java class is shown (Listing A3). In practice, the actual templates, metamodels and generated source code will be more complicated than presented in this example.

```
package {{package}};

{{#dependencies}}
import {{package}}.{{classname}};
{{/dependencies}}

public class {{classname}} extends {{supertype.classname}} {
  {{#variables}}
  {{type}} get{{name}}() {
    return getComponent("{{name}}");
  }

  {{datatype}} get{{name}}Value() {
    return get{{name}}().getValue();
  }

  void set{{name}}Value({{datatype}} value) {
    return get{{name}}().setValue(value);
  }

  {{/variables}}
  {{#methods}}
  public Variant[] call{{name}}(Variant[] inArgs) {
    // TODO: Implement method
  }

  {{/methods}}
}
```

Listing A1: The example Mustache template used in this section.

```
{
  "package": "com.example.opcua.generatedtypes",
  "classname": "ValveType"
  "dependencies":
```

```
    {
      {"package": "com.example.opcua.generatedtypes", "classname": "
          DeviceType"},
      {"package": "com.prosysopc.ua.server.nodes", "classname": "
          UaVariable"},
      {"package": "org.opcfoundation.ua.builtintypes", "classname":
          "Variant"}
    },
  "supertype": {"classname": "DeviceType"},
  "variables":
    {
      {"type": "UaVariable", "name": "IsOpen", "datatype": "Boolean"
          },
      {"type": "UaVariable", "name": "Flow", "datatype": "Double"}
    },
  "methods":
    {
      {"name": "Open"}
    }
}
```

Listing A2: The example metamodel instance used in this section, presented in Javascript Object Notation (JSON).

```
package com.example.opcua.generatedtypes;

import com.example.opcua.generatedtypes.DeviceType;
import com.prosysopc.ua.server.nodes.UaVariable;
import org.opcfoundation.ua.builtintypes.Variant;

public class ValveType extends DeviceType {
        UaVariable getIsOpen() {
                return getComponent("IsOpen");
        }

        Boolean getIsOpenValue() {
                return getIsOpen().getValue();
        }

        void setIsOpenValue(Boolean value) {
                return getIsOpen().setValue(value);
        }

        UaVariable getFlow() {
```

```
                return getComponent("Flow");
        }

        Double getFlowValue() {
                return getFlow().getValue();
        }

        void setFlowValue(Double value) {
                return getFlow().setValue(value);
        }

        public Variant[] callOpen(Variant[] inArgs) {
                // TODO: Implement method
        }

}
```

Listing A3: Resulting Java source code file.